MS-PASCAL
VOLUME I

MS-PASCAL VOLUME I

User's Guide

Introduction 7
1. Getting Started 1-1
2. A Sample Session 2-1
3. More About Compiling 3-1
4. More About Linking 4-1
5. Using a Batch Command File 5-1
6. Compiling and Linking Large Programs 6-1
7. Using Assembly Language Routines 7-1
8. Advanced Topics 8-1
Appendix A Version Specifics A-1
Appendix B MS-LINK Error Messages .. B-1

Reference Manual

Introduction 1
1. Language Overview 1-1
2. Notation 2-1
3. Identifiers 3-1
4. Introduction to Data Types..... 4-1
5. Simple Types 5-1
6. Arrays, Records, and Sets..... 6-1
7. Files 7-1
8. Reference and Other Types 8-1
9. Constants 9-1
10.Variables and Values 10-1

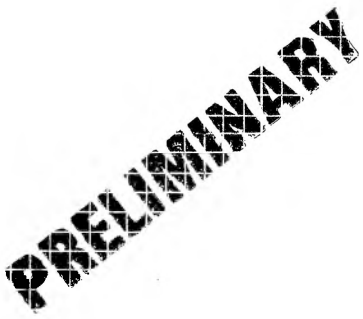
MS-PASCAL VOLUME II

Reference Manual (Continued)

11.Expressions 11-1
12.Statements 12-1
13.Introduction to Procedures and Functions 13-1

14.Available Procedures and
Functions 14-1
15.File-Oriented Procedures and
Functions 15-1
16.Compilable Parts of a Program ...16-1
17.MS-Pascal Metacommands 17-1
Appendix A. MS-Pascal Syntax
Diagrams A-1
Appendix B. MS-Pascal Features and
the ISO Standard B-1
Appendix C. MS-Pascal and Other
Pascals C-1
Appendix D. ASCII Character Codes .. D-1
Appendix E. Summary of MS-Pascal
Reserved Words E-1
Appendix F. Summary of Available
Procedures & Functions . F-1
Appendix G. Summary of MS-Pascal
Metacommands G-1
Appendix H. Messages H-1

MS-PASCAL
User's Guide



COPYRIGHT

(C) 1983 by VICTOR. (R)
(C) 1983 by Microsoft Corporation. 1983

Published by arrangement with Microsoft Corporation, whose software has been customized for use on various desktop microcomputers produced by VICTOR. Portions of the text hereof have been modified accordingly.

All rights reserved. This publication contains proprietary information which is protected by copyright. No part of this publication may be reproduced, transcribed, stored in a retrieval system, translated into any language or computer language, or transmitted in any form whatsoever without the prior written consent of the publisher. For information contact:

VICTOR Publications
380 El Pueblo Road
Scotts Valley, CA 95066
(408) 438-6680

TRADEMARKS

VICTOR is a registered trademark of Victor Technologies, Inc.
CP/M-86 is a registered trademark of Digital Research, Inc.
MS-Pascal, MS-DOS, MS-FORTRAN and MS-LINK are registered trademarks of Microsoft Corporation.

NOTICE

VICTOR makes no representations or warranties of any kind whatsoever with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. VICTOR shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this publication or its contents.

VICTOR reserves the right to revise this publication from time to time and to make changes in the content hereof without obligation to notify any person of such revision or changes.

Preliminary VICTOR release April, 1983.

ISBN 0-88182-026-1 Printed in U.S.A.

IMPORTANT SOFTWARE DISKETTE INFORMATION

For your own protection, do not use this product until you have made a backup copy of your software diskette(s). The backup procedure is described in the user's guide for your computer.

Please read the **DISKID** file on your new software diskette. **DISKID** contains important information including:

- The product name and revision number
- The part number of the product.
- The date of the DISKID file.
- A list of files on the diskette, with a description and revision number for each one.
- Configuration information (when applicable).
- Release notes giving special instructions for using the product.
- Information not contained in the current manual, including updated, additions, and deletions.

To read the **DISKID** file onscreen, follow these steps:

1. Load the operating system.
2. Remove your system diskette and insert your new software diskette.
3. Enter--

   TYPE DISKID

4. The contents of the **DISKID** file is displayed on the screen. If the file is large (more than 24 lines), the screen display will scroll. Type ALT-S to freeze the screen display; type ALT-S again to continue scrolling.

MS-PASCAL USER'S GUIDE CONTENTS

INTRODUCTION 7

System Requirements 7
Documentation 7
About This Manual 8
Manual Conventions 8
References 9

1. GETTING STARTED 1-1

1.1 Preliminary Procedures..... 1-1
1.1.1 Backing Up Your System Files... 1-1
1.1.2 Preparing Your Run-Time Library..... 1-1
1.1.3 Copying PASKEY to the Default Drive..... 1-2
1.1.4 Setting Up Your System Disk.... 1-2
1.2 Program Development..... 1-3
1.3 Vocabulary..... 1-8
1.3.1 Stages in Program Development.. 1-8
1.3.2 Linking and Run-Time..... 1-8

2. A SAMPLE SESSION 2-1

2.1 Creating a MS-Pascal Source File 2-2
2.2 Compiling Your MS-Pascal Program..... 2-3
2.2.1 Pass One..... 2-3
2.2.2 Pass Two..... 2-7
2.2.3 Pass Three..... 2-8
2.3 Linking Your MS-Pascal Program..... 2-9
2.4 Executing Your MS-Pascal Program..... 2-12

3. MORE ABOUT COMPILING 3-1

3.1 Files Written by the Compiler..... 3-1
3.1.1 Object File..... 3-1
3.1.2 Source Listing File..... 3-1
3.1.3 Object Listing File..... 3-2

3.1.4 Intermediate Files..... 3-3
3.2 Filename Conventions..... 3-4
3.3 Starting the Compiler..... 3-8
3.3.1 No Parameters on the Command Line..... 3-9
3.3.2 All Parameters on the Command Line..... 3-9
3.3.3 Some Parameters on the Command Line..... 3-11
3.4 Pass One Compiler Switches..... 3-11

4. MORE ABOUT LINKING 4-1

4.1 Files Read by the Linker..... 4-1
4.1.1 Object Modules..... 4-1
4.1.2 Libraries..... 4-3
4.2 Files Written by the Linker..... 4-5
4.2.1 The Run File..... 4-5
4.2.2 The Linker Listing File..... 4-5
4.2.3 VM.TMP..... 4-6
4.3 Linker Switches..... 4-7

5. USING A BATCH COMMAND FILE 5-1

6. COMPILING AND LINKING LARGE PROGRAMS 6-1

6.1 Avoiding Limits on Code Size..... 6-1
6.2 Avoiding Limits on Data Size..... 6-2
6.3 Working with Limits on Compile-Time Memory..... 6-4
6.3.1 Identifiers..... 6-4
6.3.2 Complex Expressions..... 6-7
6.4 Working with Limits on Disk Size..... 6-8
6.4.1 Pass One..... 6-8
6.4.2 Pass Two..... 6-10
6.4.3 Linking..... 6-11
6.4.4 A Complex Example..... 6-12
6.5 Minimizing Load Module Size..... 6-14
6.5.1 I/O..... 6-15

6.5.2 Run-Time Error Handling..... 6-16
6.5.3 Real Number Operations..... 6-16
6.5.4 Error Checking..... 6-16

7. USING ASSEMBLY LANGUAGE ROUTINES 7-1

7.1 Calling Conventions..... 7-1
7.2 Internal Representations of Data Types..... 7-3
7.2.1 Initialized Variables..... 7-8
7.3 Interfacing to Assembly Language Routines..... 7-8

8. ADVANCED TOPICS 8-1

8.1 Structure of the Compiler..... 8-1
8.1.1 The Front End..... 8-3
8.1.2 The Back End..... 8-5
8.1.2.1 Pass Two..... 8-5
8.1.2.2 Pass Three..... 8-8
8.2 An Overview of the File System..... 8-8
8.3 Run-time Architecture..... 8-12
8.3.1 Run-Time Routines..... 8-12
8.3.2 Memory Organization..... 8-13
8.3.3 Initialization and Termination..... 8-18
8.3.3.1 Machine Level Initialization..... 8-20
8.3.3.2 Program Level Initialization..... 8-22
8.3.3.3 Program Termination..... 8-24
8.3.4 Error Handling..... 8-25
8.3.4.1 Machine Error Context..... 8-28
8.3.4.2 Source Error Context..... 8-29

APPENDIX A Version Specifics A-1

A.1 Implementation Additions..... A-1
A.2 Implementation Restrictions..... A-6
A.3 Unimplemented Features..... A-7

APPENDIX B MS-LINK Error Messages B-1

FIGURES

1-1: Program Development..... 1-5
7-1: Contents of the Frame..... 7-1
7-2: Stack Before Transfer to ADD..... 7-10
7-3: One-Byte Return Value..... 7-11
7-4: Two-Byte Return Value..... 7-12
7-5: Four-Byte Return Value..... 7-12
8-1: Structure of the MS-Pascal Compiler... 8-1
8-2: Unit U Interface 8-10
8-3: Memory Organization..... 8-17

TABLES

1-1: Suggested System Disk Set Up..... 1-3
2-1: Files Used by the MS-Pascal Compiler..... 2-9
3-1: Default Filename Extensions..... 3-5
3-2: Filenames Assigned by the Compiler.... 3-5
3-3: Pass One Compiler Switches..... 3-12
4-1: Linker Defaults..... 4-3
4-2: MS-LINK Switches..... 4-7
8-1: Unit Identifier Suffixes..... 8-13
8-2: Error Code Classification..... 8-27
8-3: Run-Time Values in BRTEQQ..... 8-28

INTRODUCTION

The MS-Pascal compiler accepts programs written according to the ISO standard and programs written in the full MS-Pascal language (described in the MS-Pascal Reference Manual). This User's Guide explains how to use the MS-Pascal compiler implemented for the MS-DOS operating system.

SYSTEM REQUIREMENTS

A 256K system is required to compile MS-Pascal source code; at least 128K is required at run-time. You also need the MS-DOS operating system and the MS-LINK utility.

The current implementation of the MS-Pascal compiler can take advantage of, but does not require, an 8087 numeric coprocessor.

DOCUMENTATION

The MS-Pascal User's Guide provides an introduction to compilation and linking, a sample session, and a technical reference for the MS-Pascal compiler.

The MS-Pascal Reference Manual describes the syntax and use of the MS-Pascal language. This is the language supported by the MS-Pascal compiler, with the exceptions noted in Appendix A of the User's Guide. Any recent changes are described in the DISKID file on the distribution diskette.

ABOUT THIS MANUAL

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

Chapters 1 through 4 should be read in their entirety by the first-time user of the MS-Pascal compiler.

MANUAL CONVENTIONS

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

CAPS	Uppercase letters indicate portions of statements or commands that must be entered, exactly as shown.
< >	Angle brackets indicate user-supplied data. For lowercase text (e.g., <filename>), you supply a specific data entry of the type defined by the text. For uppercase text, press the key named by the text (such as <RETURN>).
[ ]	Square brackets indicate that the enclosed entry is optional.
...	Ellipses indicate that an entry may be repeated as many times as needed or desired.

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

Pressing the Return (or Enter) key is assumed at the end of every line you enter in response to a prompt. If a Return is the only response required, however, <RETURN> is shown.

REFERENCES

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

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

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

Jensen, Kathleen, and Wirth, Niklaus. Pascal User Manual and Report. Springer-Verlag, 1974, 1978.

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

Schneider, G.M., Weinhart, S.W., and Perlman, D.M. An Introduction to Programming and Problem Solving With Pascal. John Wiley & Sons, second edition, 1982.

1. GETTING STARTED

1.1 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 Operator's Reference Guide.

1.1.1 BACKING UP YOUR SYSTEM FILES

This step is optional but highly recommended.

The first thing you should do after you unwrap your system disks is make working copies of the disks. You can make the copies with the DCOPY utility supplied with MS-DOS. Save (archive) the original disks; if your working copies are damaged, you can make more copies from the originals.

1.1.2 PREPARING YOUR RUN-TIME LIBRARY

This step is required.

Two different run-time libraries are part of the MS-Pascal compiler software:

1. PASCAL.L87 lets you perform real number operations with an 8087 coprocessor.

2. PASCAL.LEM provides software support for real number operations.

During linking, the linker automatically searches a run-time library called PASCAL.LIB. Therefore, depending on whether or not you have the 8087

coprocessor, you must rename the appropriate system library as PASCAL.LIB (use the MS-DOS command REN).

To use PASCAL.L87, you must have an 8087 installed; programs linked with PASCAL.LEM work whether you have an 8087 or not. See Section 4.1.2 for information about how MS-LINK uses the run-time libraries.

1.1.3 COPYING PASKEY TO THE DEFAULT DRIVE

This step is required.

PASKEY, one of the files that is part of the MS-Pascal compiler, contains the MS-Pascal predeclarations. Because these predeclarations are used by the first pass of the compiler, the PASKEY file must always be on the disk in the default drive while pass one is executing.

Before you begin to compile the sample program or any program of your own, copy PASKEY to the disk in the default drive.

1.1.4 SETTING UP YOUR SYSTEM DISK

This step is recommended.

Before you begin compiling and linking a program, check the contents of your system disk against the list in Table 1-1. Make sure you have all the files you need including the linker utility, MS-LINK, from your MS-DOS package. The disk set up shown in Table 1-1 avoids reprompting from the system to reload certain MS-DOS files and eliminates the need to switch disks between passes of the compiler.

Table 1-1: Suggested System Disk Set Up

CONTENTS
COMMAND.COM
<text editor>*
<other utilities>**
PAS1.EXE
PAS2.EXE
PAS3.EXE
PASCAL.LIB
LINK.EXE

* Any text editor that fits.

** MS-DOS utilities to set up printer, clear screen, sort directory, and so on.

To prepare a system disk, first FORMAT the disk. Then use SYSCOPY to put the operating system on the disk. (The Operator's Reference Guide contains instructions for FORMAT and SYSCOPY.) If you do not put the operating system on the disk, the compiler may prompt you to reinsert your MS-DOS disk after each step, if it needs to reload COMMAND.COM. Finally, COPY the appropriate files to the disk.

1.2 PROGRAM DEVELOPMENT

This section provides a short introduction to program development (a multi-step process that includes first writing the program, and then compiling, linking, and executing it). For a brief explanation of terms that may be unfamiliar, see Section 1.3.

A microprocessor can execute only its own machine instructions; it cannot execute source program statements directly. Therefore, before you run a program, the statements in the program must be translated into the machine language of your microprocessor.

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 "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 2.2 and in Chapter 3.

A successfully compiled program must be linked before it can be executed. 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 run-time library. The linker saves your program on disk as an executable file, ready to run.

At link-time, you can link more than one object module. You can also link routines written in assembly language or other high-level languages and routines in other libraries. Linking is described in greater detail in Section 2.3 and in Chapter 4.

Figure 1-1 illustrates the entire program development process. The major steps in the process are described after the figure.

The following steps make up the program development process:

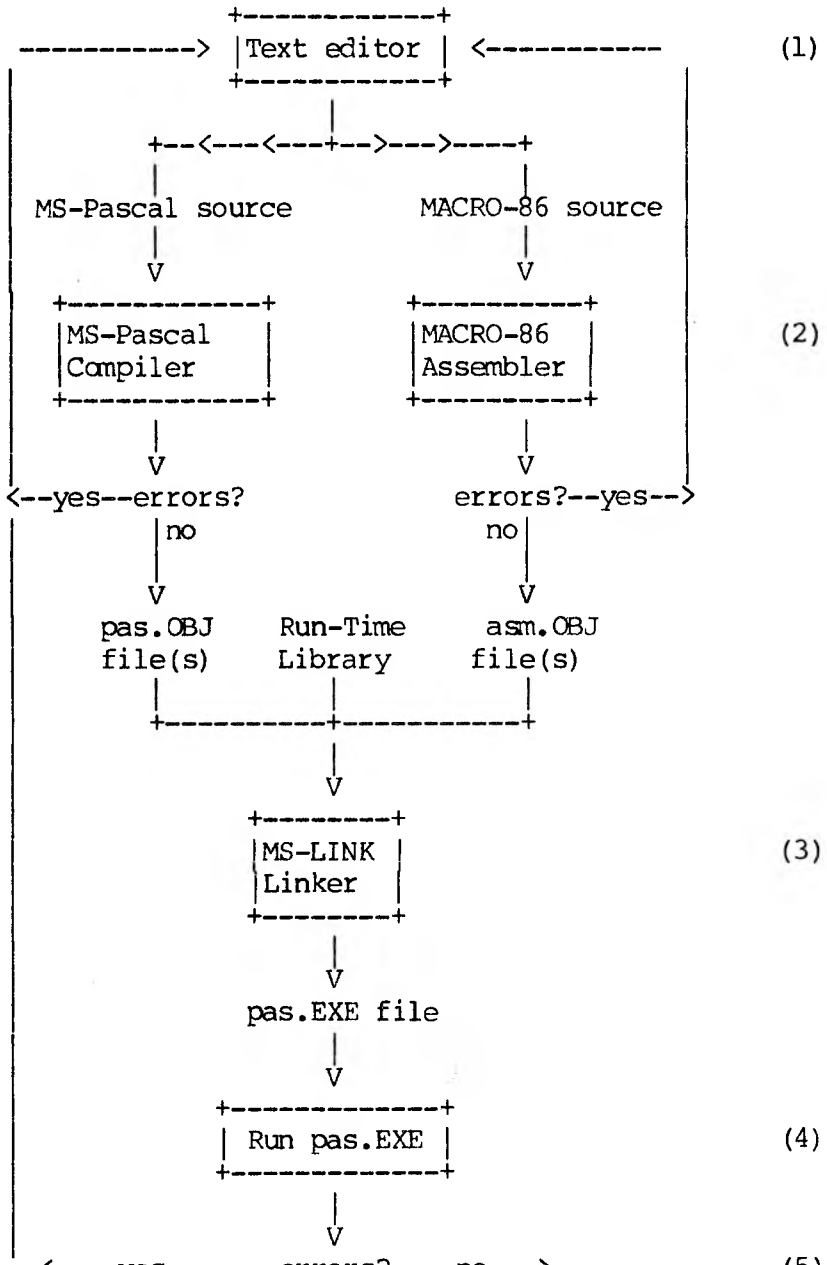
1. Create and edit MS-Pascal (and MACRO-86) source file.

   Program development begins when you write an MS-Pascal program; you can use any general purpose text editor. Also use the text editor to write any assembly language routines you plan to include.

2. Compile program with $DEBUG+. Assemble assembler source, if any.

   Once you write 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 3.4) to generate diagnostic calls for all run-time errors. If compilation is successful, the compiler creates a relocatable object file.

Figure 1-1: Program Development



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 MACRO-86 available in the Programmer's Tool Kit, Volume II.

3. Link compiled (and assembled) .OBJ files with the run-time library.

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

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 typing its filename.

5. Recompile, relink, and rerun with $DEBUG-.

   Repeat the above process until your program has successfully compiled, linked, and run without errors. Then recompile, relink, and rerun it without the run-time error-checking switches, to reduce the amount of time and space required. Chapter 6 discusses how to work within the various physical limits you may encounter in compiling, linking, and executing a program.

1.3 VOCABULARY

This section reviews some of the vocabulary used in discussing the steps in program development. The definitions are intended primarily for use with this manual. Neither the individual definitions 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 compiling, you must link the object file with the run-time library to produce an executable program or run file. The run file has the extension .EXE.

1.3.1 STAGES IN PROGRAM DEVELOPMENT

The following terms describe stages in the development and execution of a compiled program:

Compile-time: The time when the compiler is executing, and during which it compiles an MS-Pascal source file and creates a relocatable object file.

Link-time: The time when the linker is executing, during which it links together relocatable object files and library files.

Run-time: The time when a compiled and linked program is executing. By convention, run-time refers to the execution time of your program and not to the execution time of the compiler or the linker.

1.3.2 LINKING AND RUN-TIME

The following terms pertain to the linking process and the run-time library:

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 MS-Pascal Reference Manual for details. This User's Guide uses the term "module" in its general sense, unless otherwise specified.

The object files created by the compiler are "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.

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

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 run-time 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."

Relocatable module: One whose code can be loaded and run at different locations in memory. Relocatable modules contain routines and variables represented as offsets relative to the start of the module. These routines and variables are said to be at "relative" offset addresses. When the module is processed by the linker, an address is associated with the start of the module.

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

These offset addresses are still relative to a "segment," which corresponds to an 8088 segment register. Segment addresses are not defined by the linker; rather, they are computed when your program is actually loaded prior to execution.

Run-time library: Contains the run-time routines needed to implement the MS-Pascal language. A library module usually corresponds to a feature or subfeature of the MS-Pascal language.

2. SAMPLE SESSION

This chapter gives step-by-step instructions for compiling and linking an MS-Pascal program. You should 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 arises, check to see that you correctly carried out all the required procedures in Section 1.1. Then 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 run-time library.

4. Execute (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 assumes the following:

1. You have completed the necessary preliminary procedures.

2. You have two disk drives (A and B).

3. The sample program is already debugged, so that it will compile, link, and execute successfully.

4. An object listing is required. Therefore, all three passes of the compiler will be run.

5. No compiler or linker switches will be used.

6. There are no problems with data, code, or memory limits.

These assumptions are discussed in Chapters 3, 4, and 6, and are referred to as appropriate in the following sample session.

If the files required for successive steps in the process are not all on the same disk, you have to exchange disks between steps. For example, if PAS1.EXE and PAS2.EXE are not on the same disk, you 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 from pass three, you have to insert the proper system disk before running MS-LINK.

2.1 CREATING AN MS-PASCAL SOURCE FILE

Turn on your computer and load MS-DOS. Insert an empty, formatted work disk in drive B. Log on to drive B; B is now the default drive.

This sample session uses the SORT.PAS program, which came with the system software. Although you can create MS-Pascal programs with any available text editor, the source file should, in most cases, have the .PAS extension.

Copy SORT.PAS to drive B (where it would be if it were your own program). If you have not already done so, copy PASKEY to the disk in drive B. (PASKEY must always be on the default drive when you run pass one of the compiler.)

2.2 COMPILING YOUR MS-PASCAL PROGRAM

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. The sample session runs all three passes.

2.2.1 PASS ONE

Insert your Pascal system disk (see disk set up in Chapter 1) in drive A. In response to the operating system prompt, type:

A:PAS1

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

(Note that you can respond in either upper or lowercase. This manual uses uppercase simply for clarity.)

The compiler displays a sign-on message that includes the date and version number, and 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 3.

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 tells you that .PAS is the default extension for the source filename. Unless the extension is something other than .PAS, you can omit it when you type the filename.

   For this practice session, type SORT (to indicate that the source file is B:SORT.PAS).

2. Object file

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

   Object filename [SORT.OBJ]:

   The name in brackets is the name the compiler gives 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 asks 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 (no file at all). If you press the Return key, the source listing is 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 screen. Therefore, type USER in response to the source listing prompt. (Typing CON has essentially the same effect; see Section 3.2 for further information.)

4. Object listing file

The final prompt is for the object listing file, 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 are 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 respond to all four prompts. The source listing is displayed on your screen, as requested. When pass one is complete, the following message displays on your screen:

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

If the compiler detects errors during compilation, messages like the following appear instead:

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

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

- Errors are mistakes that prevent a program from running correctly.

- Warnings indicate a variety of conditions, none of which prevent the program from running, but which may reflect poor programming practice or produce invalid results.

See Appendix G in the MS-Pascal Reference Manual 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.

2.2.2 PASS TWO

Start pass two by typing:

```
A:PAS2
```

Pass two does not ordinarily prompt you for any input. It performs the following actions:

1. Reads the intermediate files PASIBF.SYM and PASIBF.BIN created in pass one.

2. Writes the object file.

3. Deletes the intermediate files created in pass one.

4. Writes two new intermediate files, PASIBF.TMP and PASIBF.OID, for use in pass three. These files are written to the 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 you do not request an object listing in pass one, pass two writes and later deletes just one new intermediate file, PASIBF.TMP.

When pass two is complete, the screen displays a message like the following example:

```
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:

- Code: The amount of space taken up by executable code.
- Cons: The amount of space taken up by constants.
- Data: The amount of space taken up by variables.

The amount of space is shown first in hexadecimal and then in decimal notation. The message giving the number of errors refers to pass two only, not to the entire compilation.

2.2.3 PASS THREE

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 2-1 summarizes the files read and written by each of the three passes of the compiler during this sample session.

Table 2-1: Files Used by the MS-Pascal Compiler

PASS	READS	WRITES	DELETES
1	SORT.PAS	USER.LST	
	PASKEY	PASIBF.SYM	
		PASIBF.BIN	
2	PASIBF.SYM	SORT.OBJ	PASIBF.SYM
	PASIBF.BIN	PASIBF.OID	PASIBF.BIN
		PASIBF.TMP	
3	PASIBF.OID	USER.COD	PASIBF.OID
	PASIBF.TMP		PASIBF.TMP

See Chapter 3 for details about compiler switches and other ways of responding to the compiler prompts.

2.3 LINKING YOUR MS-PASCAL PROGRAM

Now you are ready to link your program. Linking converts the relocatable object file into an executable program by assigning absolute addresses and setting up calls to the run-time library.

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 want to give to the executable program

3. The name you want to give to the linker listing

4. The location of the run-time library

Each of these prompts is discussed briefly in the following paragraphs and in Chapter 4. For complete information on MS-LINK, see your Programmer's Tool Kit, Volume II.

If you have not already done so, you must rename one of the run-time libraries to be PASCAL.LIB. See Section 1.1.2 for information and instructions.

1. Object modules

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

   Object Modules [.OBJ]:

   This prompt indicates that .OBJ is the default extension for any file(s) you name here. 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

   The second prompt asks 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, press the Return key.

3. Linker listing file

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

List File [NUL.MAP]:

The default for the list file is the NUL file, that is, no file at all. For this session, press the Return key to accept this default.

If, when linking your own programs, you want to display the list file on your screen, without writing it to a disk file, type CON in response to the list file prompt. (The linker does not recognize USER as a name for your console.)

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

4. Run-time library

The last linker prompt is for the location of the run-time library:

Libraries [.LIB]:

For this session, to indicate that PASCAL.LIB is on drive A, you type:

A:

After you respond to the last of the four prompts,

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

See Chapter 4 for more information on linker files and responding to the linker prompts.

2.4 EXECUTING YOUR MS-PASCAL PROGRAM

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

```
SORT
```

This command tells 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.

If the program runs correctly, you will see displayed on the screen first an unsorted list of numbers and then the same list in sorted order.

This ends the sample session. More information on compiling and on linking is provided in Chapters 3 and 4. The following listing shows a log of the entire sample session, including prompts, your responses (shown underlined), and files written to the screen (shown in brackets).

```
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]
```

3. MORE ABOUT COMPILING

This chapter gives more procedural information on the compiler, supplementing the discussion in Section 2.2. For a technical discussion of the compiler, see Section 8.1.

3.1 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 can display or print either or both of these instead of writing them to a disk file.

3.1.1 OBJECT FILE

The object file is written to disk after 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 run-time library to create an executable module containing absolute addresses.

3.1.2 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 listed in any error messages that pertain to that source line.

Compiler error messages, shown in the source listing, are also displayed on your screen. See Appendix G of the MS-Pascal Reference Manual for a list and explanation of all error messages.

Files that you include in the compilation with the $INCLUDE metacommand are also shown in the source listing. Both the $INCLUDE metacommand and the source listing are discussed in the MS-Pascal Reference Manual, Chapter 17.

The 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.

3.1.3 OBJECT LISTING FILE

The object listing file is a symbolic, assembler-like listing of the object code that 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 run-time library.

The object listing file can be useful during program development for the following reasons:

1. You can look at it 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 (see the Programmer's Tool Kit, Volume II).

3.1.4 INTERMEDIATE FILES

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

Pass two reads and then deletes PASIBF.SYM and PASIBF.BIN. Pass two itself creates one or two new intermediate files, depending on whether you 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, the following command starts pass two when the intermediate files and PAS2.EXE are on drive A:

```
A:PAS2 A/P
```

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 "/P" tells the compiler to pause before continuing so that you can insert the disk that contains the files into drive A.

After pausing, pass two prompts as follows:

Press enter key to begin pass two.

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

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

3.2 FILENAME CONVENTIONS

When you start 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.

The compiler constructs the remaining filenames from the source filename. This section describes these defaults and how to override them.

A complete MS-DOS file specification has three parts:

- **Device name:** The name of the disk drive where the file is or will be. On multi-drive machines, if you do not specify a device, the compiler assumes the logged drive.

- **Filename:** The name you give to a file. Consult your Operator's Reference Guide for any limitations on assigning filenames.

- **Filename extension:** An addition to the filename that further identifies the file. The

extension is up to three alphanumeric characters and must be preceded by a period. Although you can give any extension to a filename, the MS-Pascal compiler and MS-LINK recognize and assign certain extensions by default, as shown in Table 3-1.

Table 3-1: Default Filename Extensions

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 file
.EXE	Executable run file

If you give unique extensions to your filenames, you must include the extension as part of the filename in response to any prompt. If you do not specify an extension, the MS-Pascal compiler supplies one of those shown in Table 3-2. The last column in the table 3-2 shows the default filenames that the compiler supplies if you give a source filename and then press the Return key in response to the remaining compiler prompts.

Table 3-2: 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

The device "dev:" is the logged drive. Even if you specify a device in the source filename, the remaining file specifications default to the current logged drive. You must always 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; 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

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, use one of the special filenames USER or CON. USER is recognized by MS-Pascal (and MS-FORTRAN) only and writes to the console immediately as the listing is created. CON is recognized by all

MS-DOS programs, but saves the console output and writes it in chunks.

The general rules for filenames are summarized as follows:

1. All lowercase letters in filenames are changed to uppercase. For example, the following three names are all 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 (the default extension is .PAS). Typing ABC. instructs the compiler to accept ABC, with no extension, as the name.

3. The filename itself must not contain spaces, but leading and trailing spaces are permitted. Therefore, the following is an acceptable response to the prompt for the source filename:

   ABC ;

   The use of the semicolon is explained in rule 6.

4. You can override any defaults by typing all or part of the name instead of pressing the Return key. For example, if the logged drive is B and you want to write the object file to the disk in drive A, type A: in response to the following prompt:

   Object Filename [ABC.OBJ]:

   This creates the object file A:ABC.OBJ.

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, then the base name is the base name of the source file. Using the example from rule 4, type B: in response to the object listing prompt to give the filename 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 response 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 cannot enter a semicolon to specify a source file, since the source file has no default filename.

3.3 STARTING THE COMPILER

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

1. Let the compiler prompt you for the three filenames (as in the sample session).

2. Give all four filenames on the command line.

3. 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 5 for information.

3.3.1 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 looks like this (your responses are underlined):

```
Source filename [.PAS]: MYFILE
Object filename [MYFILE.OBJ]: <RETURN>
Source listing  [NUL.LST]: MYFILE
Object listing  [NUL.COD]: <RETURN>
```

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

Remember, pressing the Return key accepts 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.

3.3.2 ALL PARAMETERS ON THE COMMAND LINE

Instead of letting the compiler prompt you for each of the four filenames in turn, you can implicitly

or explicitly give all four names in the command line that starts the compiler. This method eliminates prompting for the filenames and is particularly useful with the MS-DOS batch files. See Chapter 5 for information on creating a batch command file for use with the compiler.

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

A:PAS1 <source>, <object>, <sourcelist>, <objectlist>;

The default naming conventions here are the ones used when filenames are prompted for.

You must separate the filenames with commas; 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 is given the source filename, 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 of the preceding command lines result in the following four filenames:

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 null listing files, use the semicolon (;) following the source filename. Thus, these command lines are equivalent:

```
A:PAS1 YOYO,YOYO,NUL,NUL;
A:PAS1 YOYO;
```

The command line can also include switches, described in Section 3.4.

3.3.3 SOME PARAMETERS ON THE COMMAND LINE

You can 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.

In the following example, you give only the names of the source file and the object file on the command line. The compiler prompts you for the names of the source listing and the object listing (your responses are underlined):

```
B: A:PAS1 TEST,TEST
Source listing [NUL.COD]: TEST
Object listing [NUL.COD]: <RETURN>
```

The preceding sequence of prompts and 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

3.4 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. You can use more than one switch, but each must begin with a slash (/). Do

not confuse these compiler switches with the linker switches, which are discussed in Section 4.3.

Switches affect the entire compilation and can be placed anywhere that spaces can go (before or after filenames, but not within them). You can enter them either on the command line or in response to compiler prompts. Table 3-3 shows the compiler switches available to you, the default position of the switch, and the corresponding metacommand.

Table 3-3: Pass One Compiler Switches

SWITCH	DEFAULT	METACOMMAND	ACTION
/A	off	$INDEXCK	Checks for array index values in range, including super array indices.
/D	off	$DEBUG	Switches on all others.
/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	on	$MATHCK	Checks for mathematical errors such as overflow and division by zero.
/N	on	$NILCK	Checks for dereferencing of any pointers that are NIL.
/Q	off	$DEBUG	Switches off all others.

/R	on	$RANGECK	Checks for subrange validity including assignments.
/S	on	$STACKCK	Checks for stack overflow at procedure or function entry.

Since all the pass one switches correspond to MS-Pascal metacommands, you can use the metacommand in the source file or give 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 17 of the MS-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 run-time 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 is spent to perform the check.

One strong caution should be observed. Use of the error-checking switches or metacommands does add considerably to the amount of code generated. Thus, you may want to use them in the early stages of program development, and later recompile your program without them to reduce code space and increase execution time.

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

This turns on $INDEXCK:

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

This turns on all the switches:

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

This sequence first turns all switches off, and then turns on $MATHCK and $INDEXCK, and later $INITCK:

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

4. LINKING

4.1 FILES READ BY THE LINKER

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

4.1.1 OBJECT MODULES

Object files can come from any of the following sources:

1. MS-Pascal compilands (programs, modules, or units)

2. MS-FORTRAN compilands (programs, subroutines, or functions)

3. User code in other high-level languages

4. Assembly language routines

5. Routines in standard run-time library modules that support facilities such as error handling, heap variable allocation, or input/output

Interfacing to MS-FORTRAN routines is straightforward. The MS-FORTRAN procedure or function must be external in the MS-Pascal source, 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 MS-Pascal or MS-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 7. For further information on MS-LINK, see the appropriate chapter in your Programmer's Tool Kit, Volume II.

Linking programs, units, and modules of MS-Pascal source code (as well as assembly language and library routines) lets you 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 6 for further information.)

For now, assume that you have created a program that uses one MS-Pascal unit and one MS-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. The files thus created are the following:

```
PROG.OBJ
UNIT.OBJ
MODU.OBJ
ASM1.OBJ
ASM2.OBJ
```

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 an MS-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 misordered. After the initial MS-Pascal object file, you can list the other modules, units, or

assembly language routines in any order.

Typing a semicolon after the name of the last object file you want to link tells the linker to omit the remaining prompts and to supply defaults, as shown in Table 4-1, for all remaining parameters.

Table 4-1: Linker Defaults

PROMPT	DEFAULT RESPONSE
Object Modules	none
Run File	prog.EXE
List Map	NUL.MAP
Libraries	PASCAL.LIB

4.1.2 LIBRARIES

A run-time library contains run-time modules required during linking to resolve references made during compilation. The MS-Pascal compiler generates space for instructions for most floating-point operations. It also issues fix-up information in the object file. During linking, these instructions are resolved using information in the run-time library.

Because MS-Pascal is designed for use on machines with or without an 8087 co-processor, the compiler provides two versions of the run-time library:

1. If you use the library PASCAL.L87 (renamed PASCAL.LIB) to link the program, the space assigned by the compiler is fixed up to become

instructions for the 8087 co-processor. The program runs correctly only with an 8087 co-processor.

2. If you use the library PASCAL.LEM (renamed PASCAL.LIB) to link your program, the instructions are transformed into emulator interrupts, which are serviced by code automatically linked in with your program. (This code is also in PASCAL.LEM.)

Because PASCAL.LIB is the only library searched automatically at link-time, you must copy or rename the library you decide to use (i.e., PASCAL.LEM or PASCAL.L87) to PASCAL.LIB. This is the only way you will be able to automatically use the kind of real number support provided by your processor. You can specify additional libraries to be searched; see your Programmer's Tool Kit, Volume II for information.

If you press the Return key in response to the final linker prompt, the linker automatically searches for a library called PASCAL.LIB on the default drive. If PASCAL.LIB is not on the default drive, the following message appears on your screen:

Cannot find library PASCAL.LIB
Enter new drive letter:

Switch disks if necessary, and then type the name of the drive that does contain PASCAL.LIB. If instead you respond by just pressing the Return key, linking proceeds without a library search. You can get the same affect by using the linker option switch /NO (short for /NODEFAULTLIBRARYSEARCH) to override the automatic search for PASCAL.LIB. This switch produces unresolved reference error messages unless you replace every required run-time 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 this prompt. See your Programmer's Tool Kit, Volume II, and complete information on using different libraries with MS-LINK.

4.2 FILES WRITTEN BY THE LINKER

The primary output of the linking process is an executable run file. You can 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.

4.2.1 THE RUN FILE

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

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

4.2.2 THE LINKER LISTING FILE

The linker map, also called the linker listing file, shows the addresses, relative to the start of the run module, for every code or data segment in your program. If you request it, with the /MAP switch,

the linker map can also include all EXTERN and PUBLIC variables. (See Section 4.3 for information on the /MAP switch.)

The linker map defaults to the null 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 want to inspect the linker map in these two instances:

- When using the debugger to set breakpoints and locate routines and variables.

- To find out why a load module is so large. (What routines are loaded? How big are they? 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 want to see the linker map without writing it 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.

4.2.3 VM.TMP

Linking begins after you respond to all the linker prompts. If the linker needs more memory space than is available, it creates a file called VM.TMP on the disk in the default drive and displays this message:

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

The linker aborts if the additional space is used up or if you remove the disk that contains VM.TMP before linking is complete.

When the linker finishes, VM.TMP is erased from the

disk, and any errors that occurred during linking are displayed. (For a list of MS-LINK error messages, see Appendix G in the MS-Pascal Reference Manual.)

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

4.3 LINKER SWITCHES

You can give one or more linker switches after any of the linker prompts. Table 4-2 summarizes the linker switches you can use with MS-Pascal. See your Programmer's Tool Kit, Volume II for more information on linker switches and when and how to use them.

Table 4-2: MS-LINK Switches

NAME	ACTION
/DSALLOCATE	Loads data at the high end of the data segment. For MS-Pascal and MS-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.

/NO — Tells the linker to not automatically search PASCAL.LIB. (Short for NODEFAULTLIBRARYSEARCH.)

/PAUSE — Tells MS-LINK to display the following message:

About to generate .EXE file
Change disks <press Return>

You can then change disks before the linker continues. The /PAUSE switch is 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: Do not use either of the additional linker switches /HIGH or /STACK with MS-Pascal and MS-FORTRAN programs.

5. USING A BATCH COMMAND FILE

The MS-DOS operating system allows you to create a batch file for executing a series of commands. Creating and using batch command files is described fully in the MS-DOS section of your Operator's Reference Guide. 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 console.

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 3.3.2.

The batch file can 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 the prompt that 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, hit Return to continue
compiling.
A:PAS2
A:LINK %1;
%1
```

To execute this file, type:

```
COLIGO SORT
```

Where: 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 hit Return if pass one was successful.

3. The third line runs pass two.

4. The fourth line links the object file.

5. The fifth line runs the executable file.

A .BAT file is executed only 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 Operator's Reference Guide.

6. COMPILING AND LINKING LARGE PROGRAMS

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

6.1 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 trade-off 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.

6.2 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 on the amount of data that can reside in the default data segment is also 64K bytes. You can go beyond this limit, however, by taking advantage of the ability to place certain kinds of data outside the default data segment, using ADS variables, 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, and so on.

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 segment, almost all MS-Pascal operations work equally well on data outside the default data segment.

The segmented data space includes the entire 8086 address space, including the default data segment. Data outside the default data segment can be referenced using ADS (segmented address) variables, VARS and CONSTS parameters, and segmented ORIGIN variables. See the MS-Pascal Reference Manual for a discussion of these MS-Pascal features.

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 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 as point #4 in "Implementation Additions" in Appendix A) 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 has a work area; a data base system has 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 attempts 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, (NEW or ALLHQQ) releases free heap blocks to the stack. Therefore, if you are running out of stack after a number of DISPOSE operations, make the following call:

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

6.3 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 this error message:

Compiler Out Of Memory

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

6.3.1 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 (PUBLIC and EXTERN) identifiers. This is good programming practice, and 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.

3. Remove unneeded identifiers from PASKEY.

   It is also possible to remove identifiers of predeclared procedures and functions you don't need from PASKEY, at least those in the final section (the one that looks like an interface). An identifier in this section must be removed three times: once from the UNIT list, once

from the interface (the declaration itself), and once from the USES list. However, the type FCBFQQ must not be removed.

You can also remove identifiers of intrinsic procedures and functions, a list near the start of PASKEY from READLN to RESULT. Any identifiers removed must be replaced with an asterisk (*). However, the procedure READFN must not be removed if you have program parameters.

Finally, the following declarations can be removed:

```
ADAPQQ      INTEGER2
ADRMEM      MAXINT
ADSMEM      MAXINT4
BYTE        MAXWORD
INTEGER1    BYTE
```

Removing any other identifiers from PASKEY generates the following error:

144 Compiler Internal Error

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. In many cases, you can declare a single interface with your global types; this is the only

interface used by other interfaces. Once your compilation gets past the USES clause in the PROGRAM, MODULE, or IMPLEMENTATION, many of these "extra" identifiers are removed.

6.3.2 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 gives 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.

6.4 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.

6.4.1 PASS ONE

For PAS1.EXE, just type PAS1 (or dev:PAS1 if necessary) to load pass one and read the PASKEY file. When the "Source File" prompt appears, you can remove the disk containing PAS1 and PASKEY. 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 non-default 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 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's can also be simply typed at your console. Just give USER as the name of your source file, and type your $INCLUDE metacommands directly, one per line. You will need to type an ALT-Z (end-of-file) to end the compilation.

If your source file doesn't fit on one diskette, 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 "Ø" to the "Inconst ZERROR" prompt, you get a compiler error. The compiler error stops the writing of the intermediate files, which leaves room on the disk for your listing. However, then you have to run the front twice, once to generate intermediate files for PAS2 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.

6.4.2 PASS TWO

Two command line parameters available with PAS2 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 /P switch tells PAS2 to pause while you remove the disk containing PAS2.EXE and insert another disk.

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

Press ENTER key to begin pass two

Take out the PAS2.EXE disk and insert the disk with the intermediate file from PAS1. Now press the Enter (Return) key, and PAS2 will run.

If you have two drives, but you run out of disk memory when executing PAS2, 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 PAS2 a drive letter to specify the drive containing the PASIBF.SYM and PASIBF.BIN files; for example, "PAS2 B". Normally you also need the pause command; for example, "PAS2 B/P". PAS2 responds 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 PAS2 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, on 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.

6.4.3 LINKING

If you are making a large program with 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 prompts 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 diskette. However, creation of the virtual file VM.TMP and the link map limit the amount of disk swapping you can do.

On a one-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 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 link process, you cannot swap the disk that gets the map.

The linker prompts 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 the link map, if you want one). Use the other drive for your object files, libraries, and run file (using the /PAUSE switch). With this method you can link very large programs.

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

6.4.4 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.

1. Pass one.

   a. Insert your MS-Pascal system disk (containing PASKEY, PAS1.EXE, PAS2.EXE and LINK.EXE) in drive A.

   b. Type PAS1, and wait for the Source File prompt.

   c. Insert the disk containing the source file LARGE.PAS in drive B.

   d. Respond to prompt with B:LARGE,B:LARGE; and wait for PAS1 to run.

2. Pass two.

   a. Type PAS2 B/P and wait for the PAS2 prompt.

   b. Remove the MS-Pascal system disk from A and insert an empty disk (to which the object file will be written).

   c. Respond to the prompt by pressing the Return key and wait for PAS2 to run.

   d. Remove the disk containing the object file from A.

3. Linking.

   a. Log on to drive B (which contains a now-empty disk).

   b. Insert your MS-Pascal system disk in A; type A:LINK and wait for Object Modules prompt.

   c. Remove the system disk from A and insert the disk containing the object file(s).

the disk containing the object file(s).

d. Respond to the prompt by typing A:LARGE (plus any other object files).

e. Respond to the Run File prompt by typing LARGE/PAUSE.

f. Respond to the List File prompt by pressing the Return key, or type B:LARGE to get a linker map.

g. Respond to the Libraries prompt by pressing the Return key or with a library name (the library must be on A).

h. Wait for linker to run, swapping the A disk after prompts as necessary.

6.5 MINIMIZING LOAD MODULE SIZE

Some 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. Run-time error messages
3. Real number operations
4. Debugging

6.5.1 I/O

Because most MS-Pascal programs perform I/O, they require linking to the MS-Pascal file system in the run-time library. However, some programs do not perform I/O and others perform I/O by directly

calling MS-Pascal's Unit U file routines or calling operating system I/O routines. For more information on Unit U, see Section 8.2.

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 still loads the Unit U procedures necessary to access the terminal (INIUQQ, ENDUQQ, PTYUQQ, PLYUQQ, and GTYUQQ), so that the runtime system can write any run-time 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.

6.5.2 RUN-TIME ERROR HANDLING

If run-time 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.

6.5.3 REAL NUMBER OPERATIONS

If an MS-Pascal program does no real number operations, it doesn't require INIX87 and ENDX87, the modules that initialize and terminate the real number support system. The dummy object module NULR7.OBJ provides dummy routines for these two modules.

6.5.4 ERROR CHECKING

Compiling and linking a program with the error-checking switches or metacommands on may generate up to 40% 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 runs considerably faster.

7. USING ASSEMBLY LANGUAGE ROUTINES

After describing the MS-Pascal calling conventions and internal representations of data types, this chapter shows how to interface 8086-88 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 13 of the MS-Pascal Reference Manual)

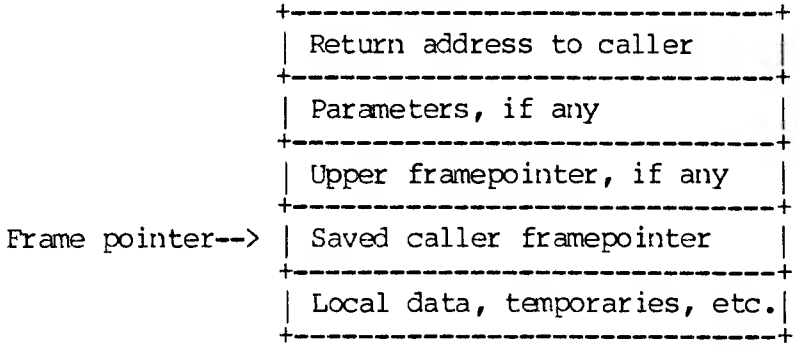
2. Procedure and function parameters (see Chapter 13 of the MS-Pascal Reference Manual)

3. MACRO-86 (or the assembler that is available for your version of MS-DOS)

7.1 CALLING CONVENTIONS

At run-time, each active procedure or function has a "frame" allocated on the stack. The frame contains the data shown in Figure 7-1.

Figure 7-1: 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 7.3.

The standard entry and exit sequences (with $STACKCK-) are as follows:

```
PUSH    BP
MOV     BP,SP
<body of routine>
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 frame temporary 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).

8086-88 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 1-byte value in AL, a 2-byte value in AX, and a 4-byte value in DX:AX (high part:low part, or segment:offset).

7.2 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 MS-FORTRAN Compiler User's Guide.

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..127) is allocated an 8-bit byte. WORD values are 16-bit unsigned numbers, but a WORD subrange in the range 0..255 is allocated an 8-bit byte. For 16-bit INTEGERs and WORDs, the least significant byte has the lower, even address.

INTEGER4 and REAL

INTEGER4s 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 fifteen 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 "not a number" (NaN).

CHAR, BOOLEAN, and Enumerated Types

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

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

Reference Types

Pointer values currently take 16 bits. A pointer is a default data segment offset. 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 point 6), 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 ON for free blocks and OFF for allocated blocks. The starting and ending heap addresses are WORD variables in BEGHQQ and ENDHQQ.

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.

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.

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 Ø 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 2nd bit (i.e., 2#01000000) of the 9th byte in a set is 1 if 'A' is in the set.

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.

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 (but this may change).

7.2.1 INITIALIZED VARIABLES

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 textfile 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, INTEGERs 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, however, are never initialized by $INITCK:

   - Variables found in a VALUE section
   - Variant fields in a record
   - Super arrays allocated on the heap

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

7.3 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 that 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 multi-dimensional super arrays, bounds are pushed in reverse order (the last flexible bound is pushed first).

As shown in Figure 7-5, 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.

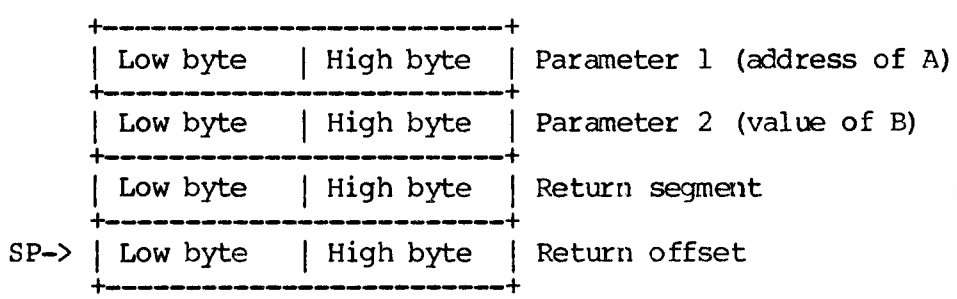
For example, assume that you have created and compiled the following program, which contains the EXTERN function ADD:

```
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 7-2.

Figure 7-2: Stack Before Transfer to ADD



Before you can run such a program, however, you 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
```

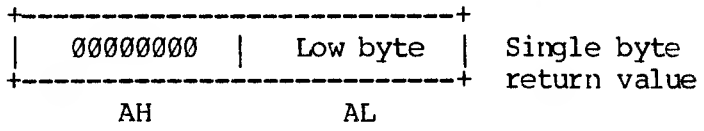
```
ADDS    SEGMENT 'CODE'
PUBLIC ADD
ADD     PROC FAR
        PUSH BP          ;Save framepointer on stack
        MOV  BP,SP       ;Address parameters
        MOV  AX,6[BP]    ;AX := value of B
        MOV  BX,8[BP]    ;BX := address of A
        ADD  AX,[BX]     ;AX := integer A + integer B
        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 7-3. AH need not be set.

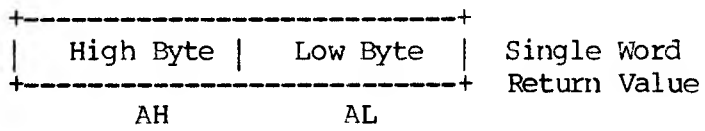
Figure 7-3: One-Byte Return Value



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.

Figure 7-4: Two-Byte Return Value

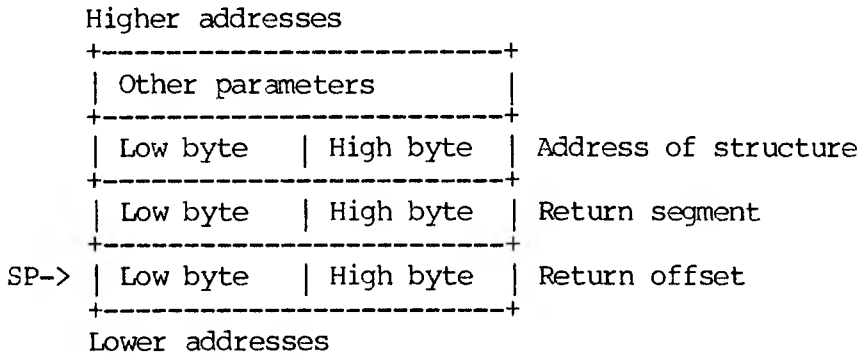


If the return value is a four-byte value, the high part (or segment) of the return value is placed in the DX register and the low part (or offset) in the AX register. (This is sometimes shown as DX:AX.) Note that this only applies to INTEGER4 and ADS types.

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 7-5. (This address is an offset, and therefore only one word is pushed.)

Figure 7-5: 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 want to pass data using PUBLIC and EXTERN variables instead of parameters. If so, these variable declarations go into a segment named DATA with classname 'DATA', in group DGROUP. It is important that you give the correct segment, class, and group names, as shown in the last example. (See your Programmer's Tool Kit, Volume II for more information.)

8. ADVANCED TOPICS

This chapter contains advanced technical information of interest primarily to experienced programmers. Since MS-FORTRAN and MS-Pascal have the same compiler back end, and share a common file and run-time system, much of the information that follows refers to both languages. Differences, where they exist, are noted.

8.1 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 8-1 illustrates the basic structure of the compiler and its relationship to the files that it reads and writes.

Pass one, which normally corresponds to a file named PAS1, is the front end of the compiler. It performs the following actions:

1. Reads the source program.

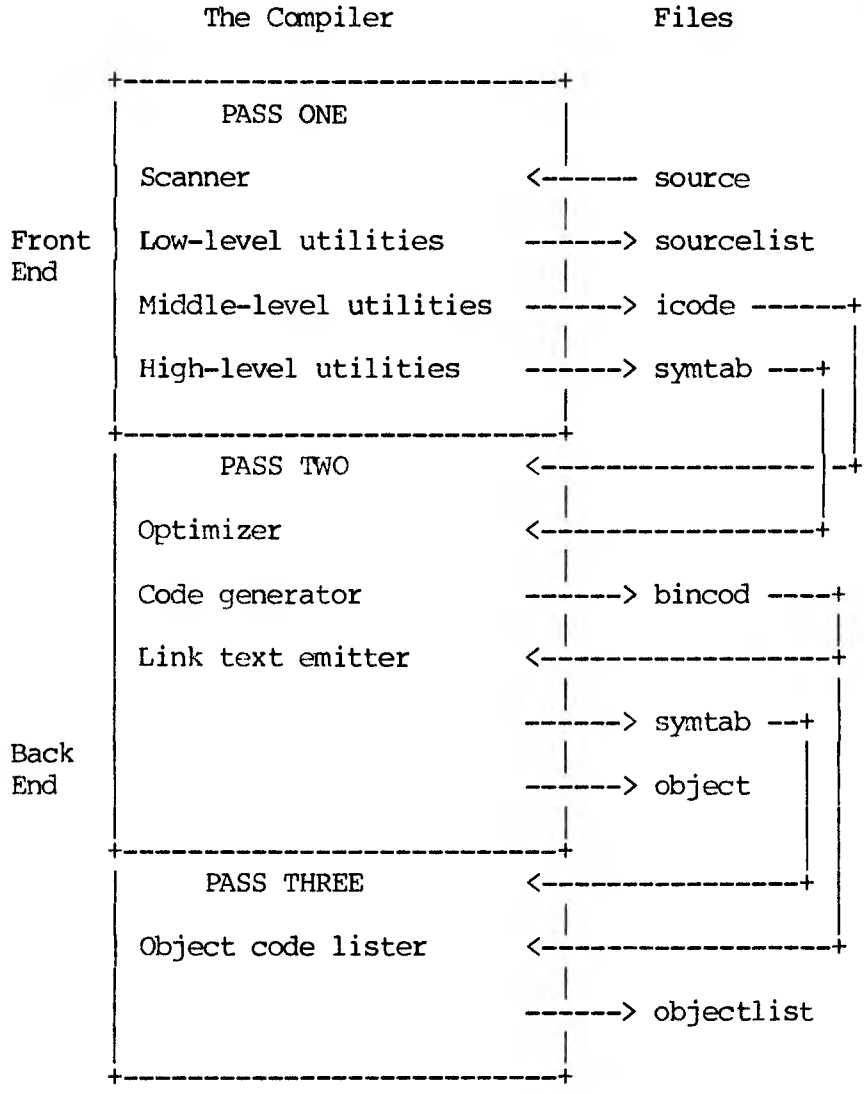
2. Compiles the source into an intermediate form.

3. Writes the source listing file.

4. Writes the symbol table file.

5. Writes the intermediate code file.

Figure 8-1: Structure of the MS-Pascal Compiler



Passes two and three (PAS2 and PAS3) together make up the back end of the compiler, which does the

following:

1. Optimizes the intermediate code.

2. Generates target code from intermediate code.

3. Writes and reads the intermediate binary file.

4. Writes the object (link text) file.

5. Writes the object listing file.

Both the front and back end of the compiler are written in MS-Pascal, in a source format that can be transformed into either relatively standard Pascal or into system level MS-Pascal. (For information on these levels, see the MS-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, and so on.

8.1.1 THE FRONT END

The MS-Pascal front end can be divided into four 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 metalanguage 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.

The front end reads a file called PASKEY to initialize pre-declared identifiers such as INTEGER, READ, and MAXINT. PASKEY can be divided into four sections:

1. The first contains the number of bytes in a file control block and the primitive type identifiers.

2. The second section lists all the intrinsic procedure and function identifiers (those that are transformed by the front end in special ways).

3. The third section contains constants, types, and external procedures and functions using normal MS-Pascal syntax.

4. The fourth contains one or more INTERFACE and USES clauses for predeclared procedures and functions.

8.1.2 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.

8.1.2.1 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 program(mer) is preserved. For instance, in the following program fragment, the array address A [J, K] must be calculated only once.

```
A [J, K] := A [J, K] + 1;
{J := J - 1;}
IF A [J, K] = MAX THEN PUNT;
```

However, if the above fragment is rewritten to include the assignment to J, shown above 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:

- Constant folding not done by the front end
- Strength reduction (changing multiplications and divisions into shifts when possible)
- 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

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 must also allocate 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.

8.1.2.2 Pass Three, The Object Code Lister

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.

8.2 AN OVERVIEW OF THE FILE SYSTEM

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

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

Each file has an associated FCB (file control block). The FCB record type begins with a number of standard fields 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 MS-Pascal user can access FCB fields directly, as explained in Chapter 7 of the MS-Pascal Reference Manual. There is no standard way to access FCB fields within MS-FORTRAN.

Both MS-Pascal and MS-FORTRAN have two special FCBs that correspond to your keyboard and screen. These two FCBs 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 Ø (or *) and are 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 MS-Pascal files, each FCB ends with the buffer variable that contains the current file component. This means that the length of an FCB in MS-Pascal is the length of its fixed portion plus the length of the buffer variable. MS-FORTRAN files do not require buffer variables, so all are of a fixed length.

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

MS-Pascal file variables can occur:

1. In static memory

2. On the stack as local variables

3. In the heap as heap variables

In MS-Pascal, generated code initializes FCBs 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.

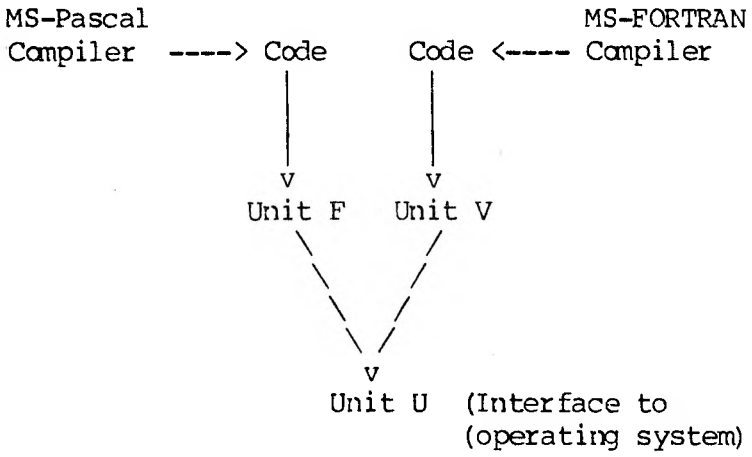
The manner of allocation and deallocation depends on the operating system. For example, a fixed number of file "slots" may be available, or the routines for MS-Pascal heap allocation may be used. In both MS-Pascal and MS-FORTRAN, an FCB can be created or destroyed, but never moved or copied.

The MS-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. This information is read by the compiler during initialization from a special file called PASKEY. The MS-FORTRAN compiler itself does not allocate files, so it doesn't need to know an FCB's length.

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

This relationship is shown schematically in Figure 8-2.

Figure 8-2: Unit U Interface



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

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

2. The fourth letter indicates a general class, where:

   a. xxxFQQ is part of the generic MS-Pascal file unit.

   b. xxxVQQ is part of the generic MS-FORTRAN file unit.

   c. xxxUQQ is part of the operating system interface unit.

File system error conditions may be 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 that Unit U has detected one, it takes one of two possible actions:

1. An immediate run-time error message is generated and the program aborts.

2. Unit F or V returns to the calling program if error trapping has been set (in MS-Pascal with the TRAP flag, in MS-FORTRAN with the ERR=nnn 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 abort 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 MS-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 environment the name is FINKXM.

8.3 RUN-TIME ARCHITECTURE

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

8.3.1 RUN-TIME ROUTINES

MS-Pascal and MS-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 8-1 shown the current unit identifier suffixes.

Table 8-1: Unit Identifier Suffixes

SUFFIX	UNIT FUNCTION
AQQ	Complex real
BQQ	Compile-time utilities
CQQ	Encode, decode
DQQ	Double precision real
EQQ	Error handling
FQQ	MS-Pascal file system
GQQ	Generated code helpers
HQQ	Heap allocator
IQQ	Generated code helpers
JQQ	Generated code helpers
KQQ	FCB definition
LQQ	STRING, LSTRING
MQQ	Reserved
NQQ	Long integer
OQQ	Other miscellaneous routines
PQQ	Pcode interpreter
QQQ	Reserved
RQQ	Real (single precision)

SQQ	Set operations
TQQ	Reserved
UQQ	Operating system file system
VQQ	MS-FORTRAN file system
WQQ	Reserved
XQQ	Initialize/Terminate
YQQ	Special utilities
ZQQ	Reserved

8.3.2 MEMORY ORGANIZATION

Memory on the 8086-88 is divided into segments, each containing up to 64K bytes. The relocatable object format and MS-LINK also put segments into classes and groups. All segments with the same class name are loaded next to each other. All segments with the same group name must reside in one area up to 64K long; that is, all segments in a group can be accessed with one segment register.

MS-FORTRAN and MS-Pascal both define a single group, named DGROUP, which is addressed using the DS or SS segment register. Normally, DS and SS contain the same value, although DS may be changed temporarily to some other segment and changed back again. SS is never changed; 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, FORTRAN blank common, and segmented addresses of FORTRAN named common blocks. The 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, blank common, and named common addresses

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, DGROUP is expanded as much as possible.

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

0000:0000 The beginning of memory on an 8086-88 system contains interrupt vectors, which are segmented addresses. Usually the first 32 to 64 are reserved for the operating system. Following these vectors is the resident portion of the operating system (MS-DOS in this case).

MS-DOS provides for loading additional code above it, which remains resident and is considered

part of the operating system as well. Examples of resident additional code are special device drivers for peripherals, a print spooler, or the debugger.

BASE:0000 — Here BASE means the starting location for loaded programs, sometimes called the transient program area. When you invoke an MS-FORTRAN or MS-Pascal program, loading begins here. The beginning of your program contains the code portion, with one or more code segments. These code segments are in the same order as the object modules given to the linker, followed by object modules loaded from libraries.

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	Addresses of named commons
CONST	CONST	Constant data
COMMQQ	COMMON	FORTRAN blank common

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

DGROUP:TOP — Here TOP means 64K bytes (4K paragraphs) above DGROUP:0000 (i.e., just past the end of DGROUP). MS-FORTRAN named common blocks start here. Each common block has a segment name as declared in the MS-

FORTRAN program as the common block name, and the class name COMMON. Each named common has one segmented (ADS) address in the COMADS segment in DGROUP. All references to common block component variables use offsets from this address.

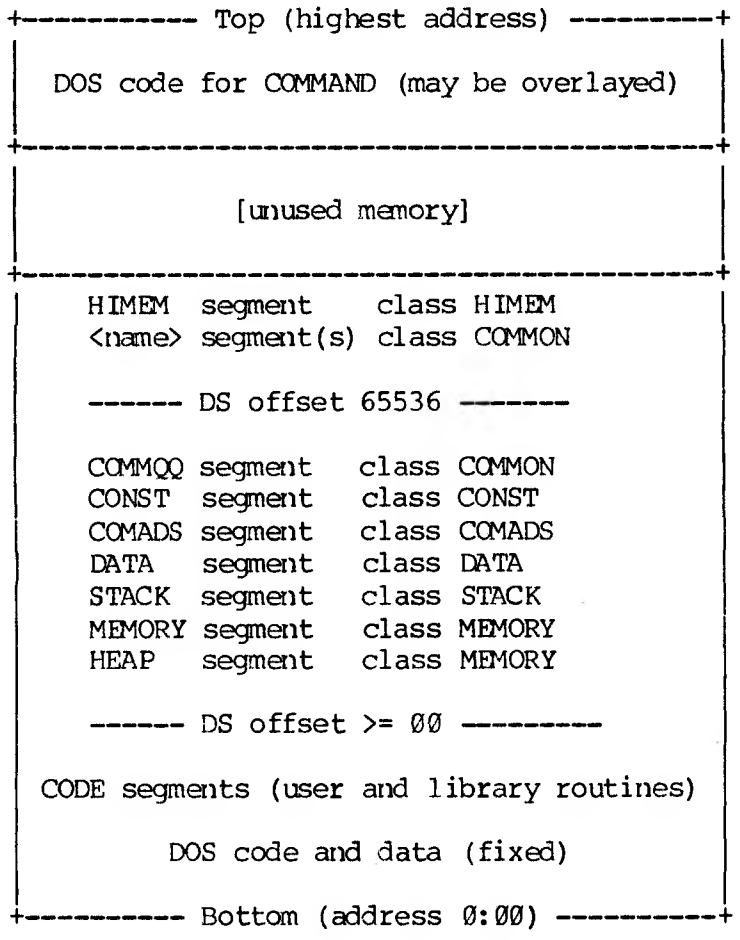
HIMEM:0000 — The segment named HIMEM (class HIMEM) gives the highest used location in the program. The segment itself contains no data, but its address is used during initialization. Available memory starts here and can be accessed with MS-Pascal ADS variables.

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

In some circumstances, the check may result 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 COMMAND.COM is on the disk in the default drive when the program ends.

Figure 8-3 illustrates this memory organization.

Figure 8-3: Memory Organization



8.3.3 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 ENTX6L. 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 a 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 MS-Pascal or MS-FORTRAN. In this case, some of the initialization and termination done by the ENTX6L module may need to be done elsewhere.

When a program is linked with the run-time library and execution begins, several levels of initialization are required. The levels, in the order in which they occur, are the following:

1. Machine-oriented initialization

2. Runtime initialization

3. Program and unit initialization

The general program structure for MS-Pascal is shown in this breakdown:

ENTX6L 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}
```

```
                    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
```

8.3.3.1 Machine Level Initialization

The entry point of an MS-Pascal load module is the routine BEGXQQ, in the module ENTXGL. BEGXQQ does the following:

1. Moves constant and static variables upward (as described in the introduction to this chapter), creating a gap for the stack and the heap. Sets the stackpointer to the top of this area.

1. Moves constant and static variables upward (as described in the introduction to this chapter), creating a gap for the stack and the heap. 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 inter-procedure GOTO to the main program.

2. Sets the framepointer to zero.

3. 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. Sets machine-dependent registers, flags, and other values.

5. 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. Calls INIX87, the real processor initializer. This routine initializes an 8087 or sets 8087 emulator interrupt vectors, as appropriate.

7. 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. Calls BEGOQQ, the escape initializer. In a normal load module, an empty BEGOQQ that just 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 run-time debugger.

9. Calls ENTGQQ, the entry point of your MS-Pascal program.

8.3.3.2 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 MS-Pascal, and MS-FORTRAN file routines are used, you must call INIVQQ to initialize the MS-FORTRAN file system. If the main program is in MS-FORTRAN, and MS-Pascal file routines are 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 or 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.

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

1. Error context initialization if $ENTRY 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 unitialized 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.

8.3.3.3 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 abort due to an error condition, either with a user call to ABORT or a run-time 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 operating system specific file unit terminator. Finally, ENDXQQ calls ENDX87 to terminate the real number processor (8087 or emulator). As with INIUQQ, INIFQQ, and INIVQQ, if your program requires no file handling, you will need to declare empty parameterless procedures for ENDYQQ and ENDUQQ.

As mentioned in Section 8.3.3, the main initialization and termination routines are in module ENTX6L. Procedures for BEGOQQ and ENDOQQ are in module MISO. ENDYQQ is in module ENDY.

8.3.4 ERROR HANDLING

Run-time errors are detected in one of four ways:

1. The user program calls EMSEQQ (i.e., ABORT).

2. A run-time 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 run-time usually involves identifying the type and location of the error and then terminating the program. The error type has three components:

- A message
- An error number
- An error status

The message describes the error, and the number can be used to look up more information (in Appendix H of the MS-Pascal Reference Manual). The message describes the error, and the number can be used to look up more information (in Appendix C of the MS-FORTRAN Reference Manual). In MS-FORTRAN, the error status value is used for special purposes and has no significance for the user. In MS-Pascal, the error status value is undefined, 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 8-2 shows the general scheme for error code numbering.

Table 8-2: 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 run-time routine (e.g., a return address). The source program context is optional; it is controlled by metacommands. If $ENTRY is on, the program context consists of:

- The source filename of the compiland containing the error
- The name of the routine in which the error occurred (program, unit, module, procedure, or function)
- The line number of the routine in the listing

file

- The page number of the routine in the listing file

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

8.3.4.1 Machine Error Context

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

The run-time entry helper, BRTEQQ, uses the run-time values shown in Table 8-3.

Table 8-3: Run-time 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 run-time routine was called from another run-time 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 run-time 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 run-time 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.

8.3.4.2 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.

APPENDIX A

VERSION SPECIFICS

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

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

A.1 IMPLEMENTATION ADDITIONS

The following additions have been made to the language described in the MS-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:

DOSXQQ (1, Ø);
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.

DOSXQQ (2, WRD ('x'));
Outputs the character 'x' to your terminal. The function return value should be ignored. The <ALT-S> command to stop and start scrolling, and the <ALT-P> command to toggle the printer, are executed if entered. Tabs are expanded.

DOSXQQ (6, 255);
Returns the next character typed on the keyboard, or zero, if no character has been typed. <ALT-S> and <ALT-P> are not treated specially. The character typed is not echoed on the terminal screen.

DOSXQQ (6, WRD ('x'));
Outputs the character 'x' to your terminal. This is the same as DOSXQQ (2, WRD ('x')), above, except that <ALT-S> and <ALT-P> are not treated specially. The function return value should be ignored in this case.

DOSXQQ (11, Ø);
Returns console status. The value 255 is returned if a character has been typed, a Ø is returned if not. This function is used to check for a keypress condition without actually reading the character.

DOSXQQ (13, Ø);
This function is not necessary in MS-DOS, but is provided for compatibility with other operating systems (such as CP/M-86), where this function resets diskette 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 Operator's Reference Guide 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 can 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 program will proceed to prompt you as follows (your responses are underlined):

```
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 halves of seconds.

6. The object code lister is no longer an integral part of pass two. If you want an object code listing, you must run the program PAS3.EXE. See Section 2.2.3 for details.

7. Four-byte functions now return values in DX:AX, not ES:BX. Also, real-valued functions now use the long return mechanism (see Section 7.1).

8. Real Number Conversion Utilities

   Releases of MS-Pascal starting with 3.0 and later 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. Microsoft to IEEE format

```
PROCEDURE M2ISQQ (VARS RMS, RIEEE:  REAL4)
```

b. IEEE to Microsoft format

```
PROCEDURE I2MSQQ (VARS RIEEE, RMS:  REAL4)
```

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

9. 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
```

A.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. MS-LINK for this version of MS-Pascal truncates global identifiers to 31 characters.

4. The PORT attribute for variables is identical to the ORIGIN attribute. It does not use I/O port addresses.

5. The maximum level to which procedures can be statically nested is 15. Dynamic nesting of procedures is limited by the size of the stack.

6. 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.

7. $SIMPLE currently turns off common subexpression optimization. $SIZE and $SPEED turn it back on (and have no other effect).

A.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. The ENABIN, DISBIN, and VECTIN library routines are not available. The INTERRUPT attribute is ignored.

5. No checking is done for invalid GOTOs.

6. READ, READLN, and DECODE cannot have M and N parameters.

7. Enumerated I/O, permitting the reading and writing of enumerated constants as strings, is not available.

8. The metacommands $TAGCK, $STANDARD, $EXTEND, and $SYSTEM can be given, but have no effect.

9. The $INCONST metacommand does not accept string constants.

APPENDIX B

MS-LINK ERROR MESSAGES

Any link error will cause the link session to abort. After you have found and corrected the problem, you must rerun MS-LINK. Link errors have no code number. See your Programmer's Tool Kit, Volume II for further information on MS-LINK.

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

There is probably a bad object file.

Bad numeric parameter

Numeric value is not in digits.

Cannot open temporary file

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

Error: dup record too complex

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

Error: fixup offset exceeds field width

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

Input file read error

There is probably a bad object file.

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

Symbol defined more than once
MS-LINK found two or more modules that define a single symbol name.

Program size or number of segments exceeds capacity of linker
The total size may not exceed 384K bytes and the number of segments may not exceed 255.

Requested stack size exceeds 64k
Specify a size greater than or equal to 64K bytes with the -STACK switch.

Segment size exceeds 64k
64K bytes is the addressing system limit.

Symbol table capacity exceeded
Very many and/or very long names were typed, exceeding the limit of approximately 25K bytes.

Too many external symbols in one module
The limit is 256 external symbols per module.

Too many groups
The limit is 10 groups.

Too many libraries specified
The limit is 8 libraries.

Too many public symbols
The limit is 1024 public symbols.

Too many segments or classes

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

Unresolved externals: <list>

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

VM read error

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

Warning: No stack segment

None of the object modules specified contains a statement allocating stack space, but you used the /STACK switch.

Warning: segment of absolute or unknown type

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

Write error in tmp file

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

Write error on run file

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

MS-PASCAL Reference Manual

PRELIMINARY

COPYRIGHT

(c) 1983 by VICTOR. (R)
(c) 1979 by Microsoft Corporation.

Published by arrangement with Microsoft Corporation, whose software has been customized for use on various desktop microcomputers produced by VICTOR. Portions of the text hereof have been modified accordingly.

All rights reserved. This publication contains proprietary information which is protected by copyright. No part of this publication may be reproduced, transcribed, stored in a retrieval system, translated into any language or computer language, or transmitted in any form whatsoever without the prior written consent of the publisher. For information contact:

VICTOR Publications
380 El Pueblo Road
Scotts Valley, CA 95066
(408) 438-6680

TRADEMARK

VICTOR is a registered trademark of Victor Technologies, Inc.
MS-DOS is a registered trademark of Microsoft Corporation.
CP/M-86 is a registered trademark of Digital Research, Inc.

NOTICE

VICTOR makes no representations or warranties of any kind whatsoever with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. VICTOR shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this publication or its contents.

VICTOR reserves the right to revise this publication from time to time and to make changes in the content hereof without obligation to notify any person of such revision or changes.

First VICTOR printing April, 1983.

ISBN: 0-88182-027-X Printed in U.S.A.

MS-PASCAL REFERENCE MANUAL CONTENTS

Introduction. 1

About This Manual. 1
Levels of MS-Pascal. 2
Selected Features. 3
Unimplemented Features 5
References 6

1. Language Overview 1-1

1.1 The Metalanguage 1-1
1.2 Programs and Compilable Parts of Programs 1-2
1.3 Procedures and Functions 1-6
1.4 Statements 1-8
1.5 Expressions 1-9
1.6 Variables 1-11
1.7 Constants 1-12
1.8 Types 1-13
1.9 Identifiers 1-14
1.10 Notation 1-15

2. Notation 2-1

2.1 Components of Identifiers 2-1
2.1.1 Letters 2-2
2.1.2 Digits. 2-2
2.1.3 The Underscore Character. 2-2
2.2 Separators 2-2
2.3 Special Symbols 2-4
2.3.1 Punctuation 2-4
2.3.2 Operators 2-6
2.3.3 Reserved Words. 2-6
2.4 Unused Characters 2-7
2.5 Notes on Characters 2-8

3. Identifiers 3-1

3.1 What is an Identifier? 3-1
3.2 Declaring an Identifier 3-3
3.3 The Scope of Identifiers 3-4
3.4 Predeclared Identifiers 3-4

4. Introduction to Data Types 4-1

4.1 What is a Type? 4-1
4.2 Declaring Data Types 4-1
4.3 Type Compatibility 4-3
4.3.1 Type Identity and Reference Parameters. 4-4
4.3.2 Type Compatibility and Expressions 4-5
4.3.3 Assignment Compatibility. 4-6

5. Simple Types. 5-1

5.1 Ordinal Types 5-1
5.1.1 INTEGER 5-2
5.1.2 WORD. 5-3
5.1.3 CHAR. 5-3
5.1.4 BOOLEAN 5-3
5.1.5 Enumerated Types. 5-4
5.1.6 Subrange Types. 5-5
5.2 Real Types 5-8
5.3 Integer4 5-11

6. Arrays, Records, and Sets 6-1

6.1 Arrays 6-1
6.2 Super Arrays 6-3
6.2.1 Strings 6-6
6.2.2 LSTRINGS 6-8
6.2.3 Using STRINGS and LSTRINGS . . . 6-10
6.3 Records 6-15
6.3.1 Variant Records 6-16

6.3.2 Explicit Field Offsets 6-19
6.4 Sets 6-21

7. Files 7-1

7.1 Declaring Files 7-1
7.2 The Buffer Variable 7-3
7.3 File Structures 7-4
7.3.1 BINARY Structure Files 7-5
7.3.2 ASCII Structure Files 7-5
7.4 File Access Modes 7-6
7.4.1 TERMINAL Mode Files 7-7
7.4.2 SEQUENTIAL Mode Files 7-8
7.4.3 DIRECT Mode Files 7-8
7.5 The Predeclared Files Input and Output . 7-9
7.6 Extend Level I/O 7-10
7.7 System Level I/O 7-12

8. Reference and Other Types 8-1

8.1 Reference Types 8-1
8.1.1 Pointer Types 8-2
8.1.2 Address Types 8-4
8.1.3 Segment Parameters for the Address Types 8-8
8.1.4 Using the Address Types 8-9
8.1.5 Notes on Reference Types 8-11
8.2 Packed Types 8-11
8.3 Procedural and Functional Types 8-12

9. Constants 9-1

9.1 What is a Constant? 9-1
9.2 Declaring Constant Identifiers. 9-2

9.3 Numeric Constants 9-3
9.3.1 REAL Constants 9-4
9.3.2 INTEGER, WORD, and INTEGER4 Constants 9-6
9.3.3 Nondecimal Numbering 9-8

9.4 Character Strings 9-8
9.5 Structured Constants 9-10
9.6 Constant Expressions 9-12

10. Variables and Values 10-1

10.1 What is a Variable? 10-1
10.2 Declaring a Variable. 10-2
10.3 The Value Section 10-3
10.4 Using Variables and Values. 10-4
10.4.1 Components of Entire Variables and Values 10-7
10.4.1.1 Indexed Variables and Values 10-7
10.4.1.2 Field Variables and Values 10-8
10.4.1.3 File Buffers and Fields . 10-9
10.4.2 Reference Variables 10-9
10.5 Attributes. 10-11
10.5.1 The STATIC Attribute 10-13
10.5.2 The PUBLIC and EXTERN Attributes 10-14
10.5.3 The ORIGIN and PORT Attributes . 10-16
10.5.4 The READONLY Attribute 10-17
10.5.5 Combining Attributes 10-18

13.4.1 Value Parameters 13-18
13.4.2 Reference Parameters 13-20
13.4.2.1 Super Array Parameters. . 13-22
13.4.2.2 Constant and Segment Parameters. 13-22
13.4.3 Procedural and Functional Parameters 13-24

14. Available Procedures and Functions . . . 14-1

14.1 Categories of Available Procedures and Functions 14-3
14.1.1 File System Procedures and Functions 14-3
14.1.2 Dynamic Allocation Procedures . 14-4
14.1.3 Data Conversion Procedures and Functions 14-4
14.1.4 Arithmetic Functions 14-5
14.1.5 Extend Level Intrinsics 14-7
14.1.6 System Level Intrinsics. 14-7
14.1.7 String Intrinsics 14-8
14.1.8 Library Procedures and Functions 14-8
14.2 Directory of Available Functions and Procedures 14-10

15. File-Oriented Procedures and Functions 15-1

15.1 File System Primitive Procedures and Functions 15-1
15.1.1 GET and PUT 15-3
15.1.2 RESET and REWRITE 15-4
15.1.3 EOF and EOLN 15-6
15.1.4 PAGE 15-7
15.1.5 Lazy Evaluation. 15-8
15.1.6 Concurrent I/O 15-10

PRELIMINARY DRAFT

15.2 Textfile Input and Output 15-12
15.2.1 READ and READLN. 15-15
15.2.2 READ Formats 15-18
15.2.3 WRITE and WRITELN 15-21
15.2.4 WRITE Formats. 15-23

15.3 Extend Level I/O. 15-28
15.3.1 Extend Level Procedures 15-28
15.3.2 Temporary Files. 15-33

16. Compilable Parts of a Program 16.1

16.1 Programs. 16-3
16.2 Modules 16-7
16.3 Units 16-9
16.3.1 The Interface Division 16-16
16.3.2 The Implementation Division 16-18

17. MS-Pascal Metacommands. 17-1

17.1 Language Level Setting and
Optimization 17-3
17.2 Debugging and Error Handling. 17-5
17.3 Source File Control 17-12
17.4 Listing File Control. 17-16
17.5 Listing File Format 17-20
17.6 Command Line Switches 17-24

Appendix A MS-Pascal Syntax Diagrams A-1

Appendix B MS-Pascal Features
and the ISO Standard. B-1
B.1 MS-Pascal and the ISO Standard. . . B-1
B.2 Summary of MS-Pascal Features . . . B-5

Appendix C MS-Pascal and Other Pascals . . . C-1
C.1 Implementations of Pascal C-1
C.2 MS-Pascal and UCSD Pascal C-4

Appendix D ASCII Character Codes D-1

Appendix E Summary of MS-Pascal
Reserved Words E-1

Appendix F Summary of Available Procedures
and Functions F-1

Appendix G Summary of MS-Pascal
Metacommands G-1

Appendix H Messages H-1

H.1 Compiler Front End Errors H-2
H.2 Compiler Back End Errors. H-39
H.3 Compiler Internal Errors. H-40
H.4 Runtime File System Errors. H-40
H.4.1 Operating System Run-time
Errors H-42
H.4.2 MS-Pascal File System
Error Codes H-44
H.5 Other Runtime Errors. H-45
H.5.1 Memory Errors H-46
H.5.2 Ordinal Arithmetic Errors. . H-48
H.5.3 Type Real Arithmetic
Errors H-49
H.5.4 Structured Type Errors . . . H-52
H.5.5 Integer4 Errors. H-53
H.5.6 Other Erors. H-53

INTRODUCTION

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

You can use MS-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 MS-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.

ABOUT THIS MANUAL

Chapter 1, "Language Overview," paints a broad picture of MS-Pascal, from the largest elements of the language down to the smallest. Later chapters build on this overview, discussing the elements one chapter at a time, starting with the smallest elements of the language and ending with a discussion of programs and other compilable units.

For information on how to use the MS-Pascal compiler and details on your specific version of MS-Pascal, see the MS-Pascal User's Guide.

PRELIMINARY DRAFT

LEVELS OF MS-PASCAL

MS-Pascal is organized into three levels: standard, extend, and system. The features of each level are discussed in more detail in Appendix B. Briefly, the differences among the three levels are as follows:

1. Standard level

   At the standard level, programs must conform to the ISO standard. Programs you create at this level are portable to and from machines running other ISO-compatible Pascal compilers. There are some minor MS-Pascal extensions to the standard that won't be caught as errors at this level. For details of these extensions, as well as other issues regarding the standard, see Appendix A. In this manual, the phrases "standard Pascal," "the ISO standard," and "at the standard level of MS-Pascal" are generally synonymous.

2. Extend level

   The extend level is intended for structured and relatively safe extensions to the ISO standard. Programs you create at this level are portable among all machines that run MS-Pascal.

3. System level

   The system level includes all features available at the extend level. It also includes some unstructured, machine-oriented extensions, such as address types and the ability to access all file control block fields, which are useful for systems programming.

In this manual, extensions to standard Pascal are

called "features." A complete list of these features and the level at which they are available are given in Appendix B. Selected features are described briefly in the following paragraphs.

In addition to these three levels, MS-Pascal has a large number of metacommands, that is, directives with which you can control the compiler. See Chapter 17 for more information.

SELECTED FEATURES

The following list includes some of the features available at the extend and system levels of MS-Pascal. For a complete list, see Section B.2, "Summary of MS-Pascal Features."

1. Underscore in identifiers, which improve readability.

2. Nondecimal numbering (hexadecimal, octal, and binary), which facilitates programming at the byte and bit level.

3. Structured constants, which you may declare in the declaration section of a program or use in statements.

4. Variable length strings (type LSTRING), as well as special predeclared procedures and functions for LSTRINGs, that overcome standard Pascal's poor string handling capabilities.

5. Super arrays, a special variable length array whose declaration permits passing arrays of different lengths to a reference parameter, as well as dynamic allocation of arrays of different lengths.

6. Predeclared unsigned BYTE (Ø-255) and WORD (Ø-65535) types that facilitate programming at the

PRELIMINARY DRAFT

system level.

7. Address types (segmented and unsegmented) that allow manipulation of actual machine addresses at the system level.

8. String reads, that allow the standard procedures READ and READLN to read strings as structures rather than character by character.

9. Interface to assembly language, provided by PUBLIC and EXTERN procedures, functions, and variables, that allows low-level interfacing to assembly language and library routines.

10. VALUE section, where you may declare the initial constant values of variables in a program.

11. Function return values of a structured type as well as of a simple type.

12. Direct (random access) files, accessible with the SEEK procedure, that enhance standard Pascal's file accessing capabilities.

13. Lazy evaluation, a special internal mechanism for interactive files that allows normal interactive input from terminals.

14. Structured BREAK and CYCLE statements, that allow structured exits from a FOR, REPEAT, or WHILE loop; RETURN statement, that allows a structured exit from a procedure or function.

15. OTHERWISE in CASE statements, whereby you avoid explicitly specifying each CASE constant. OTHERWISE also permitted with variant records.

16. STATIC attribute for variables, that allows you to indicate that a variable is to be allocated at a fixed location in memory rather than on

the stack.

17. ORIGIN attribute, that may be given to variables, procedures, and functions to indicate their absolute location in memory.

18. INTERRUPT attribute for procedures, that signals the compiler to give the procedure a special calling sequence that saves the machine status on the stack upon entry and restores the machine status upon exit.

19. Separate compilation of portions of a program (UNITs and MODULEs).

20. Conditional compilation, using conditional metacommands in your MS-Pascal source file to switch on or off compilation of parts of the source.

UNIMPLEMENTED FEATURES

The following features are either not presently implemented or implemented only as described 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. ENABIN, DISBIN, and VECTIN library routines are not available. The INTERRUPT attribute is ignored.

5. No checking is done for invalid GOTOs and uninitialized REAL values.

6. READ, READLN, and DECODE cannot have M and N parameters.

7. Enumerated I/O, for reading and writing enumerated constants as strings, is not available.

8. The metacommands $TAGCK, $STANDARD, $EXTEND, and $SYSTEM can be given, but have no effect.

9. The $INCONST metacommand does not accept string constants.

REFERENCES

The manuals in this package provide complete reference information for your implementation of the MS-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, read any of the following books:

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

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

Jensen, Kathleen, and Wirth, Niklaus. Pascal User Manual and Report. Springer-Verlag, 1974, 1978.

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

PRELIMINARY DRAFT

Schneider, G. M., Weinhart, S. W., and Perlman, D. M. An Introduction to Programming and Problem Solving With Pascal. John Wiley & Sons, second edition, 1982.

PRELIMINARY DRAFT

1. LANGUAGE OVERVIEW

This chapter gives you a summary description of MS-Pascal from the largest elements of the language down to the smallest. Each of the remaining chapters of the manual discusses these elements in detail, from the smallest element (notation) to the largest (metacommands).

1.1 METACOMMANDS

The MS-Pascal metacommands provide a control language for the MS-Pascal compiler. The metacommands let you specify options that affect the overall operation of a compilation. For example, you can conditionally compile different source files, generate a listing file, or enable or disable run-time error checking code.

All the metacommands begin with a dollar sign ($). You insert metacommands inside comment statements or give them as switches when you invoke the compiler.

Although most implementations of Pascal have some type of compiler control, the MS-Pascal metacommands are not part of standard Pascal and hence are not portable.

The metacommands available are listed in Table 1-1.

PRELIMINARY DRAFT

Table 1-1: MS-Pascal Metacommands

$BRAVE	$INTEGER	$PAGEIF	$SKIP
$DEBUG	$LINE	$PAGESIZE	$SPEED
$ENTRY	$LINESIZE	$POP	$STACKCK
$ERRORS	$LIST	$PUSH	$STANDARD
$EXTEND	$MATHCK	$RANGECK	$SUBTITLE
$GOTO	$MESSAGE	$REAL	$SYMTAB
$INCLUDE	$NILCK	$ROM	$SYSTEM
$INCONST	$OCODE	$RUNTIME	$TAGCK
$INDEXCK	$OPTBUG	$SIMPLE	$TITLE
$INTICK	$PAGE	$SIZE	$WARN
$IF $THEN $ELSE $END			

See Chapter 17 for a complete discussion of metacommands.

1.2 PROGRAMS AND COMPILABLE PARTS OF PROGRAMS

The MS-Pascal compiler processes programs, modules, and implementations of units. Collectively, these compilable programs and parts of programs are referred to as compilands. You can compile modules and implementations of units separately and then later link them to a program without having to recompile the module or unit.

The program is the fundamental unit of compilation. A program has three parts:

1. The program heading identifies the program and gives a list of program parameters.

2. The declaration section follows the program heading and contains declarations of labels, constants, types, variables, functions, and procedures. Some of these declarations are optional.

PRELIMINARY DRAFT

3. The body follows all declarations. It is enclosed by the reserved words BEGIN and END and is terminated by a period. The period is the signal to the compiler that it has reached the end of the source file.

The following program illustrates this three-part structure:

```
{Program heading}
PROGRAM FRIDAY (INPUT,OUTPUT);

{Declaration section}
LABEL 1;
CONST DAYS_IN_WEEK = 7;
TYPE  KEYBOARD_INPUT = CHAR;
VAR   KEYIN: KEYBOARD_INPUT;

{Program body}
BEGIN
   WRITE('IS TODAY FRIDAY? ');
1: READLN(KEYIN);
   CASE KEYIN OF
    'Y', 'y' : WRITELN('It''s Friday.');
    'N', 'n' : WRITELN('It''s not Friday.');
   OTHERWISE
      WRITELN('Enter Y or N.');
      WRITE('Please re-enter: ');
      GOTO 1
   END
END.
```

This three-part structure (heading, declaration section, body) is used throughout the Pascal language. Procedures, functions, modules, and units are all similar in structure to a program.

Modules are program-like units of compilation that contain the declaration of variables, constants, types, procedures, and functions, but not a program body. You can compile a module separately and later link it to a program, but it cannot be executed by itself.

Example of a module:

```
{Module heading}
MODULE MODPART;

{Declaration section}
CONST PI = 3.14;

PROCEDURE PARTA;
   BEGIN
     WRITELN ('parta')
   END;

{Body}
END.
```

A module, like a program, ends with a period. Unlike a program, a module contains no program statements.

A unit has two sections: an interface and an implementation. Like a module, an implementation can be compiled separately and later linked to the rest of the program. The interface contains the information that lets you connect a unit to other units, modules, and programs.

PRELIMINARY DRAFT

Example of a unit:

```
{Heading for interface}
INTERFACE;
UNIT MUSIC (SING, TOP);

{Declarations for interface}
VAR TOP : INTEGER;
PROCEDURE SING;

{Body of interface}
BEGIN
END;

{Heading for implementation}
IMPLEMENTATION OF MUSIC;

{Declaration for implementation}
PROCEDURE SING;
BEGIN
   FOR I := 1 TO TOP DO
   BEGIN
     WRITE ('FA '); WRITELN ('LA LA')
   END
END;

{Body of implementation}
BEGIN
   TOP := 5
END.
```

A unit, like a program or a module, ends with a period.

Modules and units let you develop large structured programs that can be broken into parts. This practice is advantageous in the following situations:

- o If a program is large, breaking it into parts makes it easier to develop, test, and maintain.

PRELIMINARY DRAFT

- If a program is large and recompiling the entire source file is time consuming, breaking the program into parts saves compilation time.

- If you intend to include certain routines in a number of different programs, you can create a single object file that contains these routines and then link it to each of the programs that uses the routines.

- If certain routines have different implementations, you can place them in a module to test the validity of an algorithm. Later you can create and implement similar routines in assembly language to increase the speed of the algorithm.

See Chapter 16 for a complete discussion of programs, modules, and units.

1.3 PROCEDURES AND FUNCTIONS

Procedures and functions act as subprograms that execute under the supervision of a main program. However, unlike programs, procedures and functions can be nested within each other and can even call themselves. Furthermore, they have sophisticated parameter-passing capabilities that programs lack.

Procedures are invoked as statements; functions can be invoked in expressions wherever values are called for.

A procedure declaration, like a program, has a heading, a declaration section, and a body.

Example of a procedure declaration:

```
{Heading}
PROCEDURE COUNT_TO(NUM : INTEGER);

{Declaration section}
VAR I : INTEGER;

{Body}
BEGIN
  FOR I := 1 TO NUM DO  WRITELN (I)
END;
```

A function is a procedure that returns a value of a particular type; hence, a function declaration must indicate the type of the return value.

Example of a function declaration:

```
{Heading}
FUNCTION ADD (VAL1, VAL2 : INTEGER): INTEGER;

{Declaration section empty}

{Body}
BEGIN
   ADD := VAL1 + VAL2
END;
```

Procedures and functions look somewhat different from programs, in that their parameters have types and other options. Like the body of a program, the body of a procedure or a function is enclosed by the reserved words BEGIN and END; however, a semicolon rather than a period follows the word "END".

Declaring a procedure or function is entirely distinct from using it in a program. For example, the procedure and function declared above might actually appear in a program as follows:

PRELIMINARY DRAFT

```
TARGET_NUMBER := ADD (5, 6);
COUNT_TO (TARGET_NUMBER);
```

See Chapter 13 for a complete discussion of procedures and functions.

See Chapters 14 and 15 for a discussion of procedures and functions that are predeclared as part of the MS-Pascal language.

1.4 STATEMENTS

Statements perform actions, such as computing, assigning, altering of the flow of control, and reading and writing files. Statements are found in the bodies of programs, procedures, and functions and are executed as a program runs. MS-Pascal statements perform the actions shown in Table 1-2.

Table 1-2: Summary of MS-Pascal Statements

STATEMENT	PURPOSE
Assignment	Replaces the current value of a variable with a new value.
BREAK	Exits the currently executing loop.
CASE	Allows for the selection of one action from a choice of many, based on the value of an expression.
CYCLE	Starts the next iteration of a loop.
FOR	Executes a statement repeatedly while a progression of values is assigned to a control variable.
GOTO	Continues processing at another part of the program.

PRELIMINARY DRAFT

IF	Together with THEN and ELSE, allows for conditional execution of a statement.
Procedure call	Invokes a procedure with actual parameter values.
REPEAT	Repeats a sequence of statements one or more times, until a Boolean expression becomes true.
RETURN	Exits the current procedure, function, program, or implementation.
WHILE	Repeats a statement zero or more times, until a Boolean expression becomes false.
WITH	Opens the scope of a statement to include the fields of one or more records, so that you can refer to the fields directly.

See Chapter 12 for a detailed discussion of each of these statements.

1.5 EXPRESSIONS

An expression is a formula for computing a value. It consists of a sequence of operators (that indicate the action to be performed) and operands (the value on which the operation is performed). Operands may contain function invocations, variables, constants, or even other expressions. In the following expression, plus (+) is an operator, while A and B are operands:

```
A + B
```


PRELIMINARY DRAFT

There are three basic kinds of expressions:

1. Arithmetic expressions perform arithmetic operations on the operands in the expression.

2. Boolean expressions perform logical and comparison operations with Boolean results.

3. Set expressions perform combining and comparison operations on sets, with Boolean or set results.

Expressions always return values of a specific type. For instance, if A, B, C, and D are all REAL variables, then the following expression evaluates to a REAL result:

```
A * B + (C / D) + 12.3
```

Expressions can also include function designators:

```
ADDREAL (2, 3) + (C / D)
```

ADDREAL is a function that has been previously declared in a program. It has two REAL value parameters, which it adds together to obtain a total. This total is the return value of the function, which is then added to (C / D).

Expressions are not statements, but can be components of statements. In the following example, the entire line is a statement; only the portion after the equal sign is an expression:

```
X := 2 / 3 + A * B
```

See Chapter 11 for a detailed discussion of expressions.

PRELIMINARY DRAFT

1.6 VARIABLES

A variable is a value that is expected to change during the course of a program. Every variable must be of a specific data type.

After you declare a variable in the heading or declaration section of a compiland, procedure, or function, it can be used in any of the following ways:

1. You can initialize it in the VALUE section of a program.

2. You can assign it a value with an assignment statement.

3. You can pass it as a parameter to a procedure or function.

4. You can use it in an expression.

The VALUE section is an MS-Pascal feature that applies only to statically allocated variables (variables with a fixed address in memory). First you declare the variables, as shown in the following example:

```
VAR I, J, K, L :  INTEGER;
```

Then you assign them initial values in the VALUE section:

```
VALUE I := 1;  J := 2;  K := 3;  L := 4;
```

Later, in statements, the variables can be assigned to and used as operands in expressions:

```
I := J + K + L;
J := 1 + 2 + 3;
K := (J * K) + 9 + (L DIV J);
```


PRELIMINARY DRAFT

See Chapter 10 for a complete discussion of variables.

1.7 CONSTANTS

A constant is a value that is not expected to change during the course of a program. At the standard level, a constant may be:

- A number, such as 1.234 and 100
- A string enclosed in single quotation marks, such as 'Miracle' or 'A1207'
- A constant identifier that is a synonym for a numeric or string constant

You declare constant identifiers in the CONST section of a compiland, procedure, or function:

```
CONST  REAL_CONST = 1.234;
       MAX_VAL    = 100;
       TITLE      = 'Pascal';
```

Because the order of declarations is flexible in MS-Pascal, you can declare constants anywhere in the declaration section of a compilable part of a program, any number of times.

Constants are closely tied to the concepts of variables and types. Variables are all of some type; types, in turn, designate a range of assumable values. These values, ultimately, are all constants.

Two powerful extensions in MS-Pascal are structured constants and constant expressions.

PRELIMINARY DRAFT

1. VECTOR, in the following example, is a structured (array) constant:

```
CONST VECTOR = VECTORTYPE (1, 2, 3, 4, 5);
```

2. MAXVAL, in the following example, is a constant expression (A, B, C, and D must also be constants):

```
CONST MAXVAL = A * (B DIV C) + D - 5;
```

See Chapter 9 for a complete discussion of these and other aspects of constants.

1.8 TYPES

Much of Pascal's power and flexibility lies in its data typing capability. Although a great variety of data types are available, they can be divided into three broad categories: simple, structured, and reference types.

1. A simple data type represents a single value, while a structured type represents a collection of values. The simple types include the following:

INTEGER	enumerated
WORD	subrange
CHAR	REAL
BOOLEAN	INTEGER4

2. The structured types include the following:

ARRAY
RECORD
SET
FILE

3. Reference types allow recursive definition of types in an extremely powerful manner.

PRELIMINARY DRAFT

All variables in Pascal must be assigned a data type. A type is either predeclared (e.g., INTEGER and REAL) or defined in the declaration section of a program. The following sample type declaration creates a type that can store information about a student:

```
TYPE
   STUDENT = RECORD
      AGE        : 5..18;
      SEX        : (MALE, FEMALE);
      GRADE      : INTEGER;
      GRADE_PT   : REAL;
      SCHEDULE   : ARRAY [1..10] OF CLASSES
   END;
```

For a detailed discussion of data types, see Chapters 4 through 8.

1.9 IDENTIFIERS

Identifiers are names that denote the constants, variables, data types, procedures, functions, and other elements of a Pascal program. Procedures and functions must have identifiers; constants, type, and variables may have identifiers (and it is useful if they do).

You, the programmer, make up most of the identifiers in a program and assign them meaning in declarations. Other identifiers are the names of variables, data types, procedures, and functions that are built into the language and need not be declared.

An identifier must begin with a letter (A - Z and a - z). The initial letter can be followed by any number of letters, digits (0 - 9), or underscore characters. The compiler ignores the case of letters; thus, "A" and "a" are equivalent.

PRELIMINARY DRAFT

The underscore in MS-Pascal is significant. Thus, the following are not identical:

FOREST

FOR_EST

The only restriction on identifiers is that you must not choose a Pascal reserved word (see Section 2.3.3 for a discussion of reserved words; see Appendix E for a complete list).

Furthermore, most compilers have some restriction either on the absolute length of an identifier or on the number of characters that are considered significant. See Appendix A in your MS-Pascal User's Guide for any limitations imposed by your version of the compiler.

See Chapter 3 for a complete discussion of identifiers in MS-Pascal.

1.10 NOTATION

The basis of all Pascal programs is an irreducible set of symbols with which the higher syntactic components of the language are created.

The underlying notation is the ASCII character set, divided into the following syntactic groups:

1. Identifiers are the names given to individual instances of components of the language.

2. Separators are characters that delimit adjacent numbers, reserved words, and identifiers.

3. Special symbols include punctuation, operators, and reserved words.

4. Some characters are not used by MS-Pascal but are available for use in a comment or string literal.

A good understanding of this notation increases your productivity by reducing the number of subtle syntactic errors in a program. See Chapter 2 for a detailed discussion of MS-Pascal notation.

PRELIMINARY DRAFT

2. NOTATION

All components of the MS-Pascal language are constructed from the standard ASCII character set. Characters make up lines that are separated by a character specific to your operating system. Lines make up files.

Within a line, individual characters or groups of characters fall into one (or more) of four broad categories:

1. Components of identifiers

2. Separators

3. Special symbols

4. Unused characters

2.1 COMPONENTS OF IDENTIFIERS

Identifiers are names that denote the constants, variables, data types, procedures, functions, and other elements of a Pascal program.

The use of identifiers is described thoroughly in Chapter 3. This section discusses only how to construct them. Identifiers must begin with a letter; subsequent components can include letters, digits, and underscore characters.

Although in theory, there is no limit on the number of characters in an identifier, most implementations restrict the length in some way. See Appendix A in your MS-Pascal User's Guide for any limitations that may apply to your system.

PRELIMINARY DRAFT

2.1.1 LETTERS

In identifiers, only the uppercase letters A through Z are significant. Although you can use lowercase letters for identifiers in a source program, the MS-Pascal compiler converts lowercase letters to uppercase.

Letters in comments or in string literals can be either uppercase or lowercase; the difference is significant. No mapping of lowercase to uppercase occurs in either comments or string literals.

2.1.2 DIGITS

Digits in Pascal are the numbers zero through nine. Digits can occur in identifiers (for example, AS129M) or in in numeric constants (for example, 1.23 and 456).

2.1.3 THE UNDERSCORE CHARACTER

The underscore (_) is the only nonalphanumeric character allowed in identifiers. The underscore is significant in MS-Pascal. Use it like a space to improve readability.

For example, the identifiers in the right column below are easier to read than those in the left hand column:

POWEROFTEN	POWER_OF_TEN
MYDOGMAUDE	MY_DOG_MAUDE

2.2 SEPARATORS

Separators delimit adjacent numbers, reserved words, and identifiers, none of which should have separators embedded within them.

A separator can be any of the following:

1. The space character

2. The tab character

3. The form feed character

4. The new line marker

5. The comment

Comments in standard Pascal take one of the following forms:

```
{This is a comment, enclosed in braces.}
(*This is an alternate form of comment.*)
```

The second form is generally used if braces are unavailable on a particular machine. Comments in either of these forms can span more than one line.

At the extend level, MS-Pascal also allows comments that begin with an exclamation point:

```
! The rest of this line is a comment.
```

For comments in this form, the new line character delimits the comment.

Nested comments are permitted in MS-Pascal, so long as each level has different delimiters. Thus, when a comment is started, the compiler ignores succeeding text until it finds the matching end-of-comment. However, such nesting may not be portable.

Always use separators between identifiers and numbers. If you fail to do so, the compiler generally issues an error or warning message. In a few cases, the MS-Pascal compiler accepts a missing separator without generating an error message.

PRELIMINARY DRAFT

For example, at extend level,

```
100MOD#127
```

is accepted as 100 MOD #127, where #127 is a hexadecimal number. However,

```
100MOD127
```

is assumed to be 100 followed by the identifier MOD127.

2.3 SPECIAL SYMBOLS

Special symbols fall into three categories:

1. Punctuation
2. Operators
3. Reserved words

2.3.1 PUNCTUATION

Punctuation in MS-Pascal serves a variety of purposes, including the those shown in Table 2-1.

PRELIMINARY DRAFT

able 2.1. Summary of Punctuation in MS-Pascal

SYMBOL	PURPOSE
{ }	Braces delimit comments.
[ ]	Brackets delimit array indices, sets, and attributes. They can also replace the reserved words BEGIN and END in a program.
( )	Parentheses delimit expressions, parameter lists, and program parameters.
'	Single quotation marks enclose string literals.
:=	The colon-equals symbol assigns values to variables in assignment statements and in VALUE sections.
;	The semicolon separates statements and declarations.
:	The colon separates variables from types, and labels from statements.
=	The equals sign separates identifiers and type clauses in a TYPE section.
,	The comma separates the components of lists.
..	The double period denotes a subrange.
.	The period designates the end of a program, indicates the fractional part of a real number, and also delimits fields in a record.

PRELIMINARY DRAFT

^	The up arrow denotes the value pointed to by a reference value.
#	The number sign denotes nondecimal numbers.
$	The dollar sign prefixes metacommands.

2.3.2 OPERATORS

Operators are a form of punctuation that indicate some operation to be performed. Some are alphabetic, others are one or two nonalphanumeric characters. Any operators that consist of more than one character must not have a separator between characters.

The operators that consist of only nonalphabetic characters are the following:

```
+   -   *   /   >   <   =   <>   <=   >=
```

Some operators (e.g., NOT and DIV) are reserved words instead of nonalphabetic characters.

See Chapter 11 for a complete list of of the nonalphabetic operators and a discussion of the use of operators in expressions.

2.3.3 RESERVED WORDS

Reserved words are a fixed part of the MS-Pascal language. They include, for example, statement names (e.g., BREAK) and words like BEGIN and END that bracket the main body of a program. See Appendix E for a complete list.

PRELIMINARY DRAFT

You cannot use a reserved word as an identifier. You can, however, declare an identifier that contains within it the letters of a reserved word (for example, the identifier DOT containing the reserved word DO).

There are several categories of reserved words in MS-Pascal:

1. Reserved words for standard level MS-Pascal

2. Reserved words added for extend level MS-Pascal features

3. Reserved words added for system level MS-Pascal features

4. Names of attributes

5. Names of directives

See Appendix E for a complete list of reserved words. The index lists where each reserved word is discussed in the manual.

2.4 UNUSED CHARACTERS

These printing characters are not used in MS-Pascal:

```
%     &     "     |     ~     `
```

You can, however, use them within comments or string literals.

The following nonprinting ASCII characters generate error messages if you use them in a source file other than in a comment or string literal:

1. The characters from CHR (Ø) to CHR (31), except the tab and form feed, CHR (9) and CHR (12), respectively

PRELIMINARY DRAFT

2. The characters from CHR (127) to CHR (255)

The tab character, CHR (9), is treated like a space and is passed on to the listing file. A form feed, CHR (12), is treated like a space and starts a new page in the listing file.

2.5 NOTES ON CHARACTERS

This section discusses special notational properties of the MS-Pascal language.

Characters within a comment or string literal are always legal and have no special effects.

Table 2-2 gives a list of pairs of printing characters that are the same ASCII character. Thus, you cannot substitute one for the other.

Table 2-2: Equivalent ASCII Characters

ASCII	PRINTS AS	EQUIVALENT CHARACTERS
CHR (94)	^	up arrow, caret
CHR (95)	_	underscore, left arrow
CHR (35)	#	number sign, English pound sign
CHR (36)	$	dollar sign, scarab (circle with four spikes)

PRELIMINARY DRAFT

MS-Pascal allows the following substitutions as well:

If your keyboard lacks:	Use this instead:
[	(.
]	.)
^	@ or ?
@	^ or ?

The substitution of a question mark (?) for an up arrow (^) is a minor extension to the ISO standard.

PRELIMINARY DRAFT

PRELIMINARY DRAFT

3. IDENTIFIERS

3.1 WHAT IS AN IDENTIFIER?

Identifiers are names for the constants, variables, data types, procedures, functions, and other elements of a Pascal program. Procedures and functions must have identifiers; constants, types, and variables may have identifiers (and it is useful if they do).

Some identifiers are predeclared; others you declare in a declaration section. Standard Pascal allows identifiers for the following elements of the Pascal language:

- Types
- Constants
- Variables
- Procedures
- Functions
- Programs
- Fields and tagfields in records

The following MS-Pascal features at the extend level also require identifiers:

- Super array types
- Modules
- Units
- Statement labels

An identifier consists of a sequence of alphanumeric characters or underscore characters. The first character must be alphabetic. Underscores in identifiers are allowed, and significant, at all levels of MS-Pascal. Two underscores in a row or an underscore at the end of an identifier are permitted.

PRELIMINARY DRAFT

Subject to the restrictions noted below, identifiers can be as long as you want. They must, however, fit on a single line. At least the first 19 characters of an identifier are significant; in some versions, as many as 31 characters are significant.

An identifier longer than the significance length generates a warning but not an error message; the excess characters are ignored by the compiler. See Appendix A in your MS-Pascal User's Guide for the significance length in your implementation.

Standard Pascal allows unsigned integers as statement labels. Statement labels have the same scope rules as identifiers (see Section 3.3). Leading zeros are not significant.

Extend level MS-Pascal allows labels that are normal alphabetic identifiers.

The identifiers for a program, module, or unit, as well as identifiers with the PUBLIC or EXTERN attribute, are passed to the linker. The operating system of a machine on which you plan to link and run a compiled MS-Pascal program may impose length restrictions on identifiers used as linker global symbols. Furthermore, the object code listing and debugger symbol table may truncate variable and procedural identifiers to six characters.

Writing programs for use with other compilers and operating systems imposes an additional constraint on a program. Such a program must conform to the identifier restrictions for the worst possible case.

For portability in general, do the following:

1. Make all identifiers unique in their first eight characters.

2. Make external identifiers unique in their first six characters.

PRELIMINARY DRAFT

3. Limit statement labels to four digits without leading zeros.

Identifiers of seven or less characters save space during compilation.

NOTE: All identifiers used internally by the run-time system are four alphabetic characters followed by the characters QQ. Avoid this form when creating new names yourself.

3.2 DECLARING AN IDENTIFIER

You declare identifiers and associate them with language objects in the declaration section of a program, module, interface, implementation, procedure, or function. Examples of identifiers, the objects they represent, and the syntax used to declare them are shown in Table 3-1. Although the details vary, the basic form of the declaration of the identifier for each of these elements is similar.

Table 3-1: Declaring Identifiers

OBJECT	IDENTIFIER	DECLARATION
Program	Z	PROGRAM Z (INPUT, OUTPUT)
Module	XXX	MODULE XXX
Interface	UUU	INTERFACE; UNIT UUU
Implementation	UUU	IMPLEMENTATION of UUU
Constant	DAYS	CONST DAYS = 365
Type	LETTERS	TYPE LETTERS = 'A'..'Z'

PRELIMINARY DRAFT

Record fields	X, Y, Z	TYPE A = RECORD X, Y ,Z : REAL END
Variable	J	VAR J : INTEGER
Label	1	LABEL 1
Label	HAWAII	LABEL HAWAII
Procedure	BANG	PROCEDURE BANG
Function	FOO	FUNCTION FOO: INTEGER

3.3 THE SCOPE OF IDENTIFIERS

An identifier is defined for the duration of the procedure, function, program, module, implementation, or interface in which you declare it. This holds true for any nested procedures or functions. An identifier's association must be unique within its scope; that is, it must not name more than one thing at a time.

A nested procedure or function can redefine an identifier only if the identifier has not already been used in it. However, the compiler does not identify such redefinition as an error, but uses the first definition until the second occurs.

A special exception for reference types is discussed in Section 8.1.5.

3.4 PREDECLARED IDENTIFIERS

PRELIMINARY DRAFT

A number of identifiers are already a part of the MS-Pascal language. This category includes the identifiers for predeclared types, super array types, constants, file variables, functions, and procedures. You can use them freely, without declaring them. However, they differ from reserved words in that you may redefine them whenever you wish.

Table 3-2 lists the predeclared identifiers in MS-Pascal.

Table 3-2: Predeclared Identifiers

STANDARD LEVEL IDENTIFIERS:

ABS	EOLN	MAXINT	PUT	SQR
ARCTAN	EXP	NEW	READ	SQRT
BOOLEAN	FALSE	ODD	READLN	SUCC
CHAR	FLOAT	ORD	REAL	TEXT
CHR	GET	OUTPUT	RESET	TRUE
COS	INPUT	PAGE	REWRITE	TRUNC
DISPOSE	INTEGER	PACK	ROUND	UNPACK
EOF	LN	PRED	SIN	WRITE
				WRITELN

STRING INTRINSICS

CONCAT	INSERT
COPYLST	POSITN
COPYSTR	SCANEQ
DELETE	SCANNE

EXTEND LEVEL INTRINSICS

ABORT	EVAL	RESULT
BYWORD	HIBYTE	SIZEOF
DECODE	LOBYTE	UPPER
ENCODE	LOWER	

SYSTEM LEVEL INTRINSICS

FILLC	MOVESL
FILLSC	MOVESR
MOVEL	RETYPE
MOVER	

EXTEND LEVEL I/O

nf	ASSIGN	READFN
CLOSE	READSET	DIRECT
SEEK	DISCARD	SEQUENTIAL
FCBFQQ	TERMINAL	FILEMODES

PRELIMINARY DRAFT

INTEGER4 TYPE

BYLONG	LOWORD
FLOATLONG	MAXINT4
HIWORD	ROUNDLONG
INTEGER4	TRUNCLONG

SUPER ARRAY TYPE

LSTRING
NULL
STRING

WORD TYPE

MAXWORD
WORD
WRD

MISCELLANEOUS

ADRMEM	INTEGER2
ADSMEM	REAL4
BYTE	REAL8
INTEGER1	SINT

PRELIMINARY DRAFT

4. INTRODUCTION TO DATA TYPES

4.1 WHAT IS A TYPE?

A type is the set of values that a variable or value can have within a program. Types are either predeclared or declared explicitly.

For example, the types INTEGER and REAL are predeclared, while the type ARRAY [1..10] OF INTEGER is declared explicitly. An explicitly declared type may also be given a type identifier. To accomplish this latter task, a type declaration is required.

Types in MS-Pascal fall into three broad categories: simple, structured, and reference types. Table 4-1 gives a breakdown of the types in each of these groups. The remainder of this chapter discusses types in general; Chapters 5 - 8 discuss the different groups in detail.

Table 4-1: Categories of Types

CATEGORY	INCLUDES	COMMENTS/EXAMPLES
Simple Types	Ordinal types	
	INTEGER	-MAXINT..MAXINT
	WORD	0..MAXWORD
	CHAR	CHR(0)..CHR(255)
	BOOLEAN	(FALSE,TRUE)
	enumerated types	e.g., (RED,BLUE)
	subrange types	e.g., 100..5000
	REAL4, REAL8	
	INTEGER4	-MAXINT4..MAXINT4

Table 4-1 (cont'd)

CATEGORY	INCLUDES	COMMENTS/EXAMPLES
Structured Types	ARRAY OF type	
	general (OF any type)	
	SUPER ARRAY (OF type)	
	STRING (n)	[1..n] of CHAR
	LSTRING (n)	[0..n] of CHAR
	RECORD	
	SET OF type	
	FILE OF	
	general (binary) files	
	TEXT	Like FILE OF CHAR
Reference Types	Pointer Types	e.g., ^TREETIP
	ADR OF type	Relative address
	ADS OF type	Segmented address
Procedural and Functional Types		only as parameter type

4.2 DECLARING DATA TYPES

The type declaration associates an identifier with a type of value. You declare types in the TYPE section of a program, procedure, function, module, interface, or implementation (not in the heading of a procedure or function).

A type declaration consists of an identifier followed by an equals sign and a type clause.

Examples of type definitions:

```
TYPE LINE = STRING (80);
     PAGE = RECORD
          PAGENUM : 1..499;
          LINES : ARRAY [1..60] OF LINE;
          FACE : (LEFT, RIGHT);
          NEXTPAGE : ^PAGE;
          END;
```

After declaring the data types, you declare variables of the types just defined in the VAR section of a program, procedure, function, module, or interface, or in the heading of a procedure or function. The following sample VAR section declares variables of the types in the preceding sample TYPE section:

```
VAR PARAGRAPH : LINE;
    BOOK : PAGE;
```

Because a type identifier is not defined until its declaration is processed by the compiler, a recursive type declaration such as the following is illegal:

```
T = ARRAY [0..9] OF T;
```

Reference types require a standard exception to this rule and are discussed in Chapter 8.

A special feature of MS-Pascal is a category called super types. A super type declaration determines the set of types that designators of that super type can assume; it also associates an identifier with the super type. Super type declarations also occur in the TYPE section. The only super types currently available in MS-Pascal are super arrays.

4.3 TYPE COMPATIBILITY

MS-Pascal follows the ISO standard for type compatibility, with some additional rules added for super array types, LSTRINGs, and constant coercions (i.e., forced changes in the type of a constant). Type transfer functions, to override the typing rules, are available with some MS-Pascal features.

Two types can be "identical," "compatible," or "incompatible." An expression may or may not be "assignment compatible" with a variable, value parameter, or array index.

4.3.1 TYPE IDENTITY AND REFERENCE PARAMETERS

Two types are identical if they have the identical identifier or if the identifiers are declared equivalent with a type definition like the following:

```
TYPE T1 = T2;
```

"Identical" types are truly identical in MS-Pascal: there is no difference between types T1 and T2 in the example above. Type identity is based on the name of the types, and not on the way they are declared or structured. Thus, for example, T1 and T2 are not identical in the following declarations:

```
TYPE T1 = ARRAY [1..10] OF CHAR;
     T2 = ARRAY [1..10] OF CHAR
```

Actual and formal reference parameters must be of identical types. Or, if a formal reference parameter is of a super array type, the actual parameter must be of the same super array type or a type derived from it. Two record or array types must be identical for assignment.

The only exception is for strings. Here, actual parameters of type CHAR, STRING, STRING (n), LSTRING, and LSTRING (n) are compatible with a formal parameter of super array type STRING.

Furthermore, an actual parameter of any FILE type may be passed to a formal parameter of a special record type FCBFQQ. Similarly, an actual parameter of type FCBFQQ may be passed to a formal parameter of any file type. See Section 7.7 for a description of the FCBFQQ type.

STRING (n) is a shorthand notation for:

PACKED ARRAY [1..n] OF CHAR

The two types are identical. However, because variables with the type LSTRING are treated specially in assignments, comparisons, READs, and WRITEs, LSTRING (n) is not a shorthand notation for PACKED ARRAY [0..n] OF CHAR. The two types are not identical, compatible, or assignment compatible. See Section 6.2.3 for further information on string types.

4.3.2 TYPE COMPATIBILITY AND EXPRESSIONS

Two simple or reference types are compatible if any of the following is true:

1. They are identical.

2. They are both ADR types.

3. They are both ADS types.

4. One is a subrange of the other.

5. They are subranges of compatible types.

Two structured types are compatible if any of the following is true:

1. They are identical.

2. They are SET types with compatible base types.

3. They are STRING derived types of equal length.

4. They are LSTRING derived types.

However, two structured types are incompatible if any of the following is true:

1. Either type is a FILE or contains a FILE.

2. Either type is a super array type.

3. One type is PACKED and the other is not.

Two values must be of compatible types when combined with an operator in an expression. (Most operators have additional limitations on the type of their operands. See Chapter 11 for details.)

A CASE index expression type must be compatible with all CASE constant values. Note that two sets are never compatible if one is PACKED and the other is not.

4.3.3 ASSIGNMENT COMPATIBILITY

Some types are implicitly compatible permitting assignment across type boundaries. For instance, assume you declare the following variables:

```
VAR DESTINATION : T_DEST;
    SOURCE      : T_SOURCE;
```

SOURCE is assignment compatible with DESTINATION (i.e., DESTINATION := SOURCE is permitted) if one of the following is true:

1. T_SOURCE and T_DEST are identical types.

2. T_SOURCE and T_DEST are compatible and SOURCE has a value in the range of subrange type T_DEST.

3. T_DEST is of type REAL and T_SOURCE is compatible with type INTEGER or INTEGER4.

4. T_DEST is of type INTEGER4 and T_SOURCE is compatible with type INTEGER or WORD.

Also, if T_DEST and T_SOURCE are compatible structured types, then SOURCE is assignment compatible with DESTINATION if one of the following is true:

1. For SETs, every member of SOURCE is in the base type of T_DEST.

2. For LSTRINGs, UPPER (DESTINATION) >= SOURCE.LEN.

Other than in the assignment statement itself, assignment compatibility is required in the following cases of implicit assignment:

- Passing value parameters
- READ and READLN procedures
- Control variable and limits in a FOR statement
- Super array type array bounds, and array indices

Assignment compatibility is usually known at compile time, and an assignment generates simple instructions. However, some subrange, set, and

LSTRING assignments depend on the value of the expression to be assigned and thus cannot be checked until run-time. If the range checking switch is on, assignment compatibility is checked at run-time; otherwise, no checking is done.

5. SIMPLE TYPES

Simple data types cannot be divided into other types, while structured types (discussed in Chapters 6 and 7) are composed of other types. The simple data types fall into three categories:

1. Ordinal types

2. REAL

3. INTEGER4

5.1 ORDINAL TYPES

Ordinal types are all finite and countable. They include the following simple types:

- INTEGER
- WORD
- CHAR
- BOOLEAN
- Enumerated types
- Subrange types

INTEGER4, though finite and countable, is not an ordinal type.

5.1.1 INTEGER

INTEGER values are a subset of the whole numbers and range from -MAXINT through Ø to MAXINT. MAXINT is the predeclared constant 32767 (i.e., 2^15 - 1) for current MS-Pascal target machines. (The value -32768 is not a valid INTEGER; the compiler uses it to check for uninitialized INTEGER and INTEGER subrange variables.)

PRELIMINARY DRAFT

INTEGER is not a subrange of INTEGER4 (discussed in Section 5.3). If it were, signed expressions would have to be calculated using the INTEGER4 type and the result converted to INTEGER.

Expressions are always calculated using a base type, not a subrange type. INTEGER type constants may be changed internally to WORD type if necessary, but INTEGER variables are not. INTEGER values change to REAL or INTEGER4 in an expression, if necessary, but not to REAL. The ORD function converts a value of any ordinal type to an INTEGER type.

The predeclared type INTEGER2 is identical to INTEGER.

5.1.2 WORD

The WORD and INTEGER types are similar, differing chiefly in their range of values. Both are ordinal types. You can think of WORD values as either a group of 16 bits or as a subset of the whole numbers from Ø to MAXWORD (65535, i.e., 2^16 - 1). The WORD type is an MS-Pascal feature that is useful in several ways:

1. To express values in the range from 32768 to 65535.

2. To operate on machine addresses.

3. To perform primitive machine operations, such as word ANDing and word shifting, without using the INTEGER type and running into the -32768 value.

Unlike INTEGERs, all WORDs are nonnegative values. The WRD function changes any ordinal type value to WORD type. Like INTEGER values, WORD values in an expression are converted to the INTEGER4 type, if necessary.

PRELIMINARY DRAFT

Having both an INTEGER and a WORD type permits mapping of 16-bit quantities in either of two ways:

1. As a signed value ranging from -32767 to +32767

2. As a positive value ranging from Ø to 65535.

However, you must not mix WORD and INTEGER values in an expression (although doing so generates a warning rather than an error message). The two assignments are not compatible.

5.1.3 CHAR

In MS-Pascal, CHAR values are 8-bit ASCII values. CHAR is an ordinal type. All 256-byte values are included in the type CHAR. In addition, SET OF CHAR is supported. Relational comparisons use the ASCII collating sequence.

Although the line-marker character used in TEXT files is not part of the CHAR type in the ISO standard, some target operating systems for MS-Pascal may require the line-marker character to be included (e.g., carriage return).

The CHR function changes any ordinal type value to CHAR type, as long as ORD of the value is in the range from Ø to 255. See Appendix D for a complete listing of the ASCII character set.

5.1.4 BOOLEAN

BOOLEAN is an ordinal type with only two (predeclared) values: FALSE and TRUE. The BOOLEAN type is a special case of an enumerated type, where ORD (FALSE) is Ø and ORD (TRUE) is 1. This means that FALSE < TRUE.

PRELIMINARY DRAFT

You may redefine the identifiers BOOLEAN, FALSE, and TRUE, but the compiler implicitly uses the former type in Boolean expressions and in IF, REPEAT, and WHILE statements.

No function exists for changing an ordinal type value to a BOOLEAN type value. However, you can achieve this effect with the ODD function for INTEGER and WORD values, or the expression:

```
ORD (value) <> 0
```

5.1.5 ENUMERATED TYPES

An enumerated type defines an ordered set of values. These values are constants and are enumerated by the identifiers that denote them.

Examples of enumerated type declarations:

```
FLAGCOLOR = (RED, WHITE, BLUE)
SUITS = (CLUB, DIAMOND, HEART, SPADE)
DOGS = (MAUDE, EMILY, BRENDAN)
```

Every enumerated type is also an ordinal type. Identifiers for all enumerated type constants must be unique within their declaration level.

At the extend level, the READ and WRITE procedures and the ENCODE and DECODE functions operate on values of an enumerated type by treating the actual constant identifier as a string. This means that enumerated values can be read directly.

The ORD function, at the standard level, can be used to change enumerated values into INTEGER values; the WRD function changes enumerated values into WORD values.

The RETYPE function, at system level, can be used to change INTEGER or WORD values to an enumerated type. For example:

```
IF RETYPE (COLOR, I) = BLUE THEN WRITELN
  ('TRUE BLUE')
```

The values obtained by applying the ORD function to the constants of an enumerated type always begin with zero. Thus, the values obtained for the type FLAGCOLOR, from the example above, are as follows:

```
ORD (RED)   = Ø
ORD (WHITE) = 1
ORD (BLUE)  = 2
```

Enumerated types are particularly useful for representing an abstract collection of names, such as names for operations or commands. Modifying a program by adding a new value to an enumerated type is much safer than using raw numbers, since any arrays indexed with the type or sets based on the type are changed automatically.

For example, interactive input of a command might be accomplished by reading the enumerated type identifier that corresponds to a command. Since enumerated types are ordered, comparisons like RED < GREEN may also be useful. At times, access to the lowest and highest values of the enumerated type is useful with the the LOWER and UPPER functions, as in the following example:

```
VAR TINT: COLOR;
FOR TINT := LOWER (TINT) TO UPPER (TINT)
   DO PAINT (TINT)
```

5.1.6 SUBRANGE TYPES

A subrange type is a subset of an ordinal type. The type from which the subset is taken is called the

PRELIMINARY DRAFT

"host" type. Therefore, all subrange types are also ordinal types.

You can define a subrange type by giving the lower and upper bound of the subrange (in that order). The lower bound must not be greater than the upper bound, but the bounds may be equal. The subrange type is frequently used as the index type of an array bound or as the base type of a set. (See Chapter 6 for a discussion of arrays and sets.)

Examples of subranges along with their host ordinal type:

```
Subrange of INTEGER:          100..200
Subrange of WORD:             WRD(1)..9
Subrange of CHAR:             'A'..'Z'
Subrange of enumerated type:  RED..YELLOW
```

In addition, you may substitute a subrange clause for a list of values in the following circumstances:

1. Set constants

2. Set constructors

3. CASE statement constants and record variant labels (at the extend level)

Besides using the subrange type in array and set declarations, you can use it to help to guarantee that the value of a variable is within acceptable bounds. If the range checking switch is on during compilation, these bounds are checked at run-time.

For instance, if the logic of a program implies that a variable always has a value from 100 to 999, then declaring it with a subrange causes the compiler to check that the variable is never assigned a value outside this range.

PRELIMINARY DRAFT

In addition, declaring a subrange type may permit the compiler to allocate less room and use simpler operations. For example, declaring BOTTLES to be the INTEGER subrange 1..1ØØ means that the type can be allocated in eight bits instead of sixteen.

Three subrange types are predeclared:

1. BYTE = WRD(Ø)..255;
   {8-bit WORD subrange}

2. SINT = -127..127;
   {8-bit INTEGER subrange}

3. INTEGER1 = SINT

The BYTE type is particularly useful in machine-oriented applications. For example, the ADRMEM and ADSMEM types (see Section 8.1.2 for details) normally treat memory as an array of bytes. However, since the BYTE type is really a subrange of the WORD type, expressions with BYTE values are calculated using 16-bit instead of 8-bit arithmetic, if necessary.

In some cases (for example, an assignment of a BYTE expression to a BYTE variable when the math checking switch is off), the compiler can optimize 16-bit arithmetic to 8-bit arithmetic. In general, using BYTE instead of WORD saves memory at the expense of BYTE-to-WORD conversions in expression calculations.

At the extend level, subrange bounds can be constant expressions. Because the compiler assumes that the left parenthesis always starts an enumerated type declaration, the first expression in a subrange declaration must not start with a left parenthesis. For example:

PRELIMINARY DRAFT

```
TYPE {First two are permitted.}
     FEE = (A, B, C);
     FIE = M + 2 * N .. (P - 2) * N;
     {FOO is invalid as declared.}
     FOO = (M + 2) * N .. P - 2 * N;
```

5.2 REAL

REAL values are nonordinal values of a given range and precision; the range of allowable values depends on the target system. The MS-Pascal User's Guide gives more specific information about your system.

Most MS-Pascal implementations use either the MS-or IEEE single precision real number format. These formats have a 24-bit mantissa and an 8-bit exponent, giving about seven digits of precision and a maximum value of 1.7Ø1411E38. MS-format REAL constants are limited to the range 1.ØE-38 to 1.ØE+38.

The current version of MS-Pascal includes expanded numeric data types for processing higher precision real (and integer) numbers. For reals, this includes support for single and double precision real numbers according to the IEEE floating-point standard.

Standard Pascal provides a type REAL. MS-Pascal provides three real types: REAL, REAL4, and REAL8. However, the type REAL is always identical to either REAL4 or REAL8. The choice is made with a metacommand, $REAL:n, where n is either 4 or 8. {$REAL:8} has the same effect as TYPE REAL = REAL8. The default type for REAL is normally REAL4, but can be changed (see Appendix A in the MS-Pascal User's Guide for details).

PRELIMINARY DRAFT

Any or all of these real number forms can be used in a single program. However, programs that use REAL4 and REAL8 are not portable.

The REAL4 type is in 32-bit IEEE format, and the REAL8 type is in 64-bit IEEE format. The IEEE standard format is as follows:

REAL4: Sign bit, 8-bit binary exponent with bias of 127, 23-bit mantissa.

REAL8: Sign bit, 11-bit binary exponent with bias of 1023, 52-bit mantissa.

In both cases the mantissa has a "hidden" most significant bit (always one) and represents a number greater than or equal to 1.0 but less than 2.0. An exponent of zero means a value of zero, and the maximum exponent means a value called NaN (not a number). Bytes are in "reverse" order; the lowest addressed byte is the least significant mantissa byte.

The REAL4 numeric range is barely seven significant digits (24 bits), with an exponent range of E-38 to E+38. The REAL8 numeric range includes over fifteen significant digits (53 bits), with an exponent range of E-306 to E+306 (a very large number!)

The exponent character can be "D" or "d" as well as "E" or "e", so a number like 12.34d56 is permitted. This minor extension provides compatibility with other MS-languages. However, the D or d exponent character does not indicate double precision (as it does in FORTRAN), since this would imply that numbers with an E or e exponent character are single precision.

REAL literals in MS-Pascal are converted first to REAL8 format and then to REAL4 as necessary (for example, to be passed as a CONST parameter or to initialize a variable in a VALUE section). If you

PRELIMINARY DRAFT

need actual REAL4 constants, you must declare them as REAL4 variables (perhaps adding the READONLY attribute) and assign them a constant in a VALUE section.

Both REAL4 and REAL8 values are passed to intrinsic functions as reference (CONSTS) parameters, rather than as value parameters. The compiler accepts REAL expressions as CONSTS parameters; it evaluates the expression, assigns the result to a stack temporary, and passes the address of the temporary, which is usually more efficient than passing the value itself (especially in the REAL8 case).

Functions that return REAL values use the long return method; that is, the caller passes an additional, hidden, offset address of a stack temporary which receives the result. This applies to all functions returning REAL4 or REAL8 values, both user-defined and intrinsic.

The inclusion of special "not-a-number" (NaN) values means that a comparison between two real numbers can have a result other than less-than, equal, or greater-than. The numbers can be unordered, meaning one or both are NaNs. An unordered result is the same as "not equal, not less than, and not greater than."

For example, if variables A or B are NaN values:

1. A < B is false.

2. A <= B is false.

3. A > B is false.

4. A >= B is false.

5. A = B is false.

6. A <> B is, however, true.

REAL comparisons do not follow all the same rules as other comparisons. A < B is not always the same as NOT (B <= A); this fact prevents some optimizations otherwise done by the compiler. If A is a NaN, then A <> A is true; in fact, this expression is a good way to check for a NaN value.

The MS-Pascal run-time library provides additional REAL functions to support MS-FORTRAN. These functions are available in MS-Pascal, but are not predeclared (see Chapter 14 for further information on the functions available and how to use them.)

5.3 INTEGER4

Like INTEGER and WORD values, INTEGER4 values are a subset of the whole numbers. INTEGER4 values range from -MAXLONG to MAXLONG. MAXLONG is a predeclared constant with the value of 2,147,483,647 (i.e., 2^31 - 1). The value -2,147,487,648 (i.e., -2^31) is not a valid INTEGER4.

Unlike INTEGER and WORD, the INTEGER4 type is not considered an ordinal type. There are no INTEGER4 subranges and INTEGER4 cannot be an array index or the base type of a set. Also, INTEGER4 values cannot be used to control FOR and CASE statements.

INTEGER4 is currently an extended numeric type, like REAL. Values of type INTEGER or WORD in an expression change automatically to INTEGER4 if the expression requires an intermediate value out of the range of either INTEGER or WORD. Values of type INTEGER4 do not change to REAL in an expression; you must explicitly use the FLOATLONG function to make the conversion.

PRELIMINARY DRAFT

PRELIMINARY DRAFT

6. ARRAYS, RECORDS, AND SETS

A structured type is composed of other types. The components of structured types are either simple types or other structured types. A structured type is characterized by the types of its components and by its structuring method. In MS-Pascal, a structured type can occupy up to 65534 bytes of memory.

The structured types in MS-Pascal are the following:

```
ARRAY <range> OF <type>
SUPER ARRAY <range> OF <type>
   STRING  (n)
   LSTRING (n)
RECORD
SET OF <base-type>
FILE OF <type>
```

Because components of structures can be structured types themselves, you may have, for example, an array of arrays, a file of records containing sets, or a record containing a file and another record. This data typing flexibility gives Pascal linguistic power as a computing language.

The remainder of this chapter discusses arrays, records, and sets. See Chapter 7 for a discussion of files.

6.1 ARRAYS

An array type is a structure that consists of a fixed number of components. All the components are of the same type (called the "component type").

The elements of the array are designated by indices, which are values of the "index type" of the array. The index type must be an ordinal type: BOOLEAN,

PRELIMINARY DRAFT

CHAR, INTEGER, WORD, subrange, or enumerated.

Arrays in Pascal are one dimensional. Since the component type can also be an array, however, n-dimensional arrays are supported as well.

Examples of type declarations for arrays:

```
TYPE
INT_ARRAY : ARRAY [1..10] OF INTEGER;
ARRAY_2D  : ARRAY [0..9] OF ARRAY [0..99] OF 0..999;
MORAL_RAY : ARRAY [PEOPLE] OF (GOOD, EVIL)
```

In the last declaration, PEOPLE is a subrange type, while GOOD and EVIL are enumerated constants. A short-hand notation available for n-dimensional arrays makes the following statement the same as the second example above:

```
ARRAY_2D :  ARRAY [0..9, 0..99] OF 0..999;
```

After declaring these arrays, you could assign components of the arrays with statements such as these:

```
INT_ARRAY [10]   := 1234;
ARRAY_2D [0,99] := 999;
MORAL_RAY [Machiavelli] := EVIL;
```

All of an n-dimensional PACKED array is packed; therefore these statements are equivalent:

```
PACKED ARRAY [1..2, 3..4] OF REAL
```

```
PACKED ARRAY [1..2] OF PACKED ARRAY [3..4] OF REAL
```

See Chapter 8 for a discussion of packed types.

PRELIMINARY DRAFT

6.2 SUPER ARRAYS

A super array is one example of an MS-Pascal "super type." A super type is like a set of types or like a function that returns a type. Super types in general, and super arrays in particular, are features of MS-Pascal.

The super array type has several important uses. In particular, you may use them for any of the following purposes:

1. To process strings. Both STRING and LSTRING are predeclared super array types. The LSTRING type handles variable length strings. STRING handles fixed-length strings and strings more than 255 characters long.

2. To dynamically allocate arrays of varying sizes. Otherwise such arrays would need a maximum possible size allocation.

3. As the formal parameter type in a procedure or function. By making such a declaration, you make the procedure or function usable for a set or class of types, rather than for just a single fixed-length type.

A super type identifier specifies the set of types represented by the super type. A later type declaration may declare a normal type identifier as a type "derived" from that class of types. This derived type is like any other type.

A super array type declaration is an array type declaration prefixed with the keyword SUPER. Every array upper bound is replaced with an asterisk (*), as shown:

```
TYPE VECTOR = SUPER ARRAY [1..*] OF REAL;
```


PRELIMINARY DRAFT

Following the above type declaration, you could declare the following variables:

```
VAR   ROW: VECTOR (10);
      COL: VECTOR (30);
      ROWP: ^ VECTOR;
```

In this example, VECTOR is a super array type identifier. VECTOR (10) and VECTOR (30) are type designators denoting "derived types." ROW and COL are variables of types derived from VECTOR. ROWP is a pointer to the super array type VECTOR.

Although the general concept of super types allows other "types of types," such as super subranges and super sets (in addition to super arrays), super types currently allow only an array type with parametric upper bounds. A super type is a class of types and not a specific type. Thus, in the VAR section of a program, procedure, or function, you cannot declare the variables to be of a super type; you must declare them as variables of a type derived from the super type.

However, a formal reference parameter in a procedure or function can be given a super type. For this reason the routine can operate on any of the possible derived types. (This kind of parameter is called a "conformant array" in other Pascals.)

A pointer referent type can also be given a super type. This allows a pointer to refer to any of the possible derived types. A pointer referent to a super type allows "dynamic arrays." These arrays are allocated on the heap by passing their upper bound to the procedure NEW. See Chapter 8 for a discussion of pointer types and dynamic allocation. See Chapter 14 for a description of the procedure NEW.

Example using the NEW procedure for dynamic allocation:

PRELIMINARY DRAFT

```
VAR STR_PNT: ^SUPER PACKED ARRAY [1..*] OF CHAR;
    VEC_PNT: ^SUPER ARRAY [Ø..*, Ø..*] OF REAL;
      .
      .
    NEW (STR_PNT, 12);
    NEW (VEC_PNT, 9, 99);
```

An actual parameter in a procedure or function can be of a super type rather than a derived type, but only if the parameter is a reference parameter or pointer referent. (These are the only kinds of variables that can be of a super rather than a derived type.)

Example of super arrays:

```
TYPE VECTOR = SUPER ARRAY [1..*] OF REAL;
{"VECTOR" is the super array type identifier.}
VAR X: VECTOR (12); Y: VECTOR (24); Z: VECTOR (36);
{X, Y, and Z are types derived from VECTOR.}

{Below, SUM accepts variables of all types}
{derived from the super type VECTOR.}
FUNCTION SUM (VAR V: VECTOR): REAL;
{V is the formal reference parameter of the
{super type VECTOR.}

VAR S: REAL; I: INTEGER;
BEGIN
  S := Ø;
  FOR I := 1 TO UPPER (V) DO S := S + V [I];
  SUM := S;
END;

BEGIN
  .
  .
  TOTAL := SUM (X) + SUM (Y) + SUM (Z);
  .
  .
END
```


PRELIMINARY DRAFT

The normal type rules for components of a super array type and for type designators that use a super array type allow components to be assigned, compared, and passed as parameters.

The UPPER function returns the actual upper bound of a super array parameter or referent. The maximum upper bound of a type derived from a super array type is limited to the maximum value of the index type implied by the lower bound (e.g., MAXINT, MAXWORD). Two super array types are predeclared, STRING and LSTRING. The compiler directly supports STRING and LSTRING types in the following ways:

1. LSTRING and STRING assignment

2. LSTRING and STRING comparison

3. LSTRING and STRING READs

4. Access to the length of a STRING with the UPPER function

5. Access to maximum length of an LSTRING with the UPPER function

6. Access to LSTRING length with STR.LEN and STR[Ø]

These subjects are discussed in Section 6.2.3.

6.2.1 STRINGS

STRINGs are predeclared super arrays of characters:

```
TYPE STRING = SUPER PACKED ARRAY [1..*] OF CHAR;
```

A string literal such as 'abcdefg' automatically has the type STRING (n). The size of the array 'abcdefg' is 7; thus, the constant is of the STRING

derived type, STRING (7).

Standard Pascal calls any packed array of characters with a lower bound of one a "string" and permits a few special operations on this type (such as comparison and writing) that you cannot do with other arrays.

In MS-Pascal, the super array notation STRING (n) is identical to PACKED ARRAY [1..n] OF CHAR (n may range from 1 to MAXINT). There is no default for n, as in some other Pascals, since STRING means the super array type itself and not a string with a default length.

The identifier STRING is for a super array, so you can only use it as a formal reference parameter type or pointer referent type. The other super array restrictions apply: you can not compare such a parameter or dereferenced pointer or assign it as a whole.

Any variable (or constant) with the super array type STRING, or one of the types CHAR or STRING (n) or PACKED ARRAY [1..n] OF CHAR, can be passed to a formal reference parameter of super array type STRING. Furthermore, a variable of type LSTRING or LSTRING (n) can also be passed to a formal reference parameter of type STRING. For a discussion of STRING as a formal reference parameter, see Section 6.2.3.

Standard Pascal supports the assigning, comparing, and writing of STRINGs. The extend level permits reading STRINGs, including the super array type STRING and a derived type STRING (n). Reading a STRING causes input of characters until the end of a line or the end of the STRING is reached. If the end of the line is reached first, the rest of the STRING is filled with blanks. Writing a string writes all of its characters.

The normal Pascal type compatibility rules are relaxed for STRINGs. Any two variables or constants with the type PACKED ARRAY [1..n] OF CHAR or the type STRING (n) can be compared or assigned if the lengths are equal. However, since the length of a STRING super array type may vary, comparisons and assignments are not allowed.

Example of an illegal STRING assignment:

```
PROCEDURE CANNOT_DO (VAR S : STRING);
VAR STR : STRING (10);
BEGIN
  STR := S
  {This assignment is illegal because}
  {the length of S may vary.}
END;
```

The PACKED prefix in the declaration PACKED ARRAY [1..n] OF CHAR (as defined in the ISO standard) normally implies that a component cannot be passed as a reference parameter. In MS-Pascal, this restriction does not apply.

To keep conformance to the ISO standard, this passing of the CHAR component of a STRING as a reference parameter is defined as an "error not caught." Also, the index type of a string is officially INTEGER, but WORD type values can also be used to index a STRING. Many string-processing applications are expected to take advantage of the LSTRING type, described in Section 6.2.2.

A number of intrinsic procedures and functions for strings are discussed in Chapter 14. Many of the procedures and functions described work on STRINGs; some apply only to LSTRINGs.

6.2.2 LSTRINGS

The LSTRING feature in MS-Pascal allows variable-

PRELIMINARY DRAFT

length strings. LSTRING (n) is predeclared as:

TYPE LSTRING = SUPER PACKED ARRAY [Ø..*] OF CHAR

However, a variable with the explicit type PACKED ARRAY [Ø..n] OF CHAR is not "identical" to the type LSTRING (n) even though they are structurally the same. There is no default for n; the range of n is from zero to 255. Characters in an LSTRING can be accessed with the usual array notation.

Internally, LSTRINGs contain a length (L), followed by a string of characters. The length is contained in element zero of the LSTRING and can vary from Ø to the upper bound. The length of an LSTRING variable T can be accessed as T[Ø] with type CHAR, or as T.LEN with type BYTE. String constants of type CHAR or STRING (n) are changed automatically to type LSTRING.

The predeclared constant NULL is the empty string, LSTRING (Ø). NULL is the only constant with type LSTRING; there is no way to define other LSTRING constants. As with STRINGs, a CHAR component of an LSTRING can be passed as a reference parameter, and WORD and INTEGER values can be used to index an LSTRING.

Several operations work differently on LSTRINGs than on STRINGs. Any LSTRING can be assigned to any other LSTRING, so long as the current length of the right side is not greater than the maximum length of the left side. Similarly, an LSTRING can be passed as a value parameter to a procedure or function, so long as the current length of the actual parameter is not greater than the maximum length specified by the formal parameter. If the range checking switch is on, the compiler checks the assignment of LSTRINGs and the passing of LSTRING (n) parameters. The actual number of bytes assigned or passed is the minimum of the upper bounds of the LSTRINGs.

Neither side in an LSTRING assignment can be a parameter of the super array type LSTRING; both must be types derived from it.

Examples of LSTRING assignments:

```
{Declaring the variables}
VAR A : LSTRING (19);
    B : LSTRING (14);
    C : LSTRING (6);
    .
    .
{Assigning the variables}
A := '19 character string';
B := '14 characters';
C := 'shorty';
A := B;
{This is legal, since the length of B}
{is less than the maximum length of A.}
C := A;
{This is illegal, since length of A}
{is greater than the maximum length of C.}
```

You may compare any two LSTRINGs, including super array type LSTRINGs (the only super array type comparison allowed). Reading an LSTRING variable causes input of characters, until the end of the current line or the end of the LSTRING, and sets the length to the number of characters read. Writing from an LSTRING writes the current length string.

6.2.3 USING STRINGS AND LSTRINGS

This section describes the STRING and LSTRING operations directly supported by the compiler. An annotated program at the end of this section illustrates the use of STRINGs and LSTRINGs in context.

Chapter 14 describes the following string procedures and functions:

CONCAT	INSERT	COPYSTR	SCANEQ
COPYLST	POSITN	DELETE	SCANNE

At the system level of MS-Pascal, the procedures FILLC, FILLSC, MOVEL, MOVESL, MOVER, and MOVESR also operate on strings.

MS-Pascal supports STRINGs and LSTRINGs directly in the following ways:

Assignment

You can assign any LSTRING value to any LSTRING variable, as long as the maximum length of the target variable is greater than or equal to the current length of the source value and neither is the super array type LSTRING. If the maximum length of the target is less than the current length of the source, only the target length is assigned, and a run-time error occurs if the range checking switch is on. You can assign a STRING value to a STRING variable, as long as the length of both sides is the same and neither side is the super array type STRING. Passing either STRING or LSTRING as a value parameter is much like making an assignment.

Comparison

LSTRING operators (< <= > >= <> =) use the length byte for string comparisons; the operands can be of different lengths. Two strings must be the same length to be considered equal. If two strings of different lengths are equal up to the length of the shorter one, the shorter is considered less than the longer one. The operands can be of the super array type LSTRING. For STRINGs, the same relational operators are available, but the lengths must be the same, and operands of the super array type STRINGs are not allowed.

PRELIMINARY DRAFT

READs and WRITEs

READ LSTRING reads until the LSTRING is filled or until the end-of-line is found. The current length is set to the number of characters read. WRITE LSTRING uses the current length. See also READSET (Chapter 15), which reads into an LSTRING as long as input characters are in a given SET OF CHAR. READ STRING pads with spaces if the line is shorter than the STRING. WRITE STRING writes all the characters in the string. Both READ and WRITE permit the super array types STRING and LSTRING, as well as their derived types.

Length access

You can access the current length of an LSTRING variable T with T.LEN, which is of type BYTE, or with T[Ø], which is of type CHAR. This notation can assign a new length, as well as determine the current length. The UPPER function finds the maximum length of an LSTRING or the length of a STRING. This is especially useful for finding the upper bound of a super array reference parameter or pointer referent.

You cannot assign or compare mixed STRINGs and LSTRINGs, unless the STRING is constant. You can assign STRINGs to LSTRINGs, or vice versa, with one of the move routines or with the COPYSTR and COPYLST procedures. Since constants of type STRING or CHAR change automatically to type LSTRING if necessary, LSTRING constants are considered normal STRING constants. NULL (the zero length LSTRING) is the only explicit LSTRING constant.

In the sample program at the end of this section, all STRING parameters (CONST or VAR) can take either a STRING or an LSTRING; all LSTRING parameters are VAR LSTRING and must take an LSTRING variable.

A "special transformation" lets you pass an actual LSTRING parameter to a formal reference parameter of type STRING. The length of the formal STRING is the actual length of the LSTRING. Therefore, if LSTR (in the following example) is of type LSTRING (n) or LSTRING, it can be passed to a procedure or function with a formal reference parameter of type STRING:

```
VAR LSTR : LSTRING (10);
  .
  .
PROCEDURE TIE_STRING (VAR STR : STRING);
  .
  .
TIE_STRING (LSTR);
```

In this case, UPPER (STR) is equivalent to LSTR.LEN.

Procedures and functions with reference parameters of super type STRING can operate equally well on STRINGs and LSTRINGs. The only reason to declare a parameter of type LSTRING is when the length must be changed. Normally, an LSTRING is either a VAR or a VARS parameter in a procedure or function, since a CONST or CONSTS parameter of type LSTRING cannot be changed.

Example of a program that uses STRINGs and LSTRINGs:

```
PROGRAM STRING_SAMPLE;

PROCEDURE STRING_PROC  (CONST S:  STRING);  BEGIN END;
PROCEDURE LSTRING_PROC (CONST S: LSTRING);  BEGIN END;

VAR
  CHR1VAR: CHAR;
  STR5VAR: STRING (5);
  LST5VAR: LSTRING (5);
  LST9VAR: LSTRING (9);
  STR4VAR: PACKED ARRAY [1..4] OF CHAR;
  STR6VAR: PACKED ARRAY [1..6] OF CHAR;
```


PRELIMINARY DRAFT

```
BEGIN

{Look at all the kinds of strings a}

{CONST STRING parameter takes.}
STRING_PROC ('A');
{Character constant is OK.}
STRING_PROC (CHR1VAR);
{Character variable is OK.}
STRING_PROC ('STRING');
{STRING constant is OK.
STRING_PROC (STR5VAR);
{STRING variable is OK.}
STRING_PROC (LST5VAR);
{LSTRING variable is OK.}

{However, a CONST LSTRING parameter cannot take}
{non-LSTRING variables.}
LSTRING_PROC ('A');
{Character constant is OK.}
LSTRING_PROC (CHR1VAR);
{Character variable is not OK!}
LSTRING_PROC ('STRING');
{STRING constant is OK.}
LSTRING_PROC (STR5VAR);
{STRING variable is not OK!}
LSTRING_PROC (LST5VAR);
{LSTRING variable is OK.}

{Assignments to a STRING variable are limited to}
{to the same type.}
STR5VAR := 'A';
{Character constant is not OK!}
STR5VAR := CHR1VAR;
{Character variable is not OK!}
STR5VAR := 'TINY';
{STRING constant too small.}
STR5VAR := 'RIGHT';
{Both sides have five characters; OK.}
STR5VAR := 'longer';
{Not OK; STRING constant is too large.}
STR5VAR := LST5VAR;
```


PRELIMINARY DRAFT

```
{Not OK; you cannot assign LSTRINGs to STRINGs.}
COPYSTR (LST5VAR, STR5VAR);
{COPYSTR is an intrinsic procedure.}
STR5VAR := STR4VAR;
{Not OK; STRING variable is too small.}
COPYSTR (STR4VAR, STR5VAR);
{COPYSTR is OK; padding of space in STR5VAR[5].}
STR5VAR := STR5VAR;
{OK; both sides have five characters.}
STR5VAR := STR6VAR;
{Not OK; STRING variable is too large.}

{Assignments to an LSTRING variable, however,}
{are more flexible.}
LST5VAR := 'A';

{Character constant is OK.}
LST5VAR := CHR1VAR;
{Character variable is not OK!}
LST5VAR := 'TINY';
{Smaller STRING constant is OK.}
LST5VAR := 'RIGHT';
{Same length STRING constant is OK.}
LST5VAR := 'LONGER';
{This gives an error at run-time only; OK for now.}
LST5VAR := LST9VAR;
{This may give an error at run-time; OK for now.}
LST9VAR := LST5VAR;
{This isn't even checked at run-time; always OK.}
LST5VAR := STR5VAR;
{Not OK; you cannot assign a STRING variable to an}
{LSTRING variable.}
COPYLST (STR5VAR, LST5VAR);
{This is the way to copy a STRING variable to an LSTRING.}

END.
```

6.3 RECORDS

A record structure acts as a template for conceptually related data of different types. The

PRELIMINARY DRAFT

record type itself is a structure consisting of a fixed number of components, usually of different types.

Each component of a record type is called a field. The definition of a record type specifies the type and an identifier for each field within the record. Because the scope of these "field identifiers" is the record definition itself, they must be unique within the declaration. The field values associated with field identifiers are accessible with record notation or with the WITH statement.

For example, you could declare the following record type:

```
TYPE LP = RECORD
            TITLE : LSTRING (100);
            ARTIST : LSTRING (100);
            PLASTIC : ARRAY
              [1..SONG_NUMBER] OF SONG_TITLE
            END
```

You could then declare a variable of the type LP, as follows:

```
VAR BEATLES_1 :  LP;
```

Finally, you could access a component of the record with either field notation or the WITH statement (note the period separating field identifiers):

```
BEATLES_1.TITLE := 'Meet The Beatles';
WITH BEATLES_1 DO
   PLASTIC[1] := 'I Wanna Hold Your Hand'
```

6.3.1 VARIANT RECORDS

A record can have several "variants," in which case a certain field called the "tag field" indicates which variant to use. The tag field may or may not

have an identifier and storage in the record. Some operations, such as the NEW and DISPOSE procedures and the SIZEOF function, can specify a tag value even if the tag is not stored as part of the record.

Examples of variant records:

```
TYPE OBJECT = RECORD
       X, Y: REAL;
       CASE S: SHAPE OF
         SQUARE: (SIZE, ANGLE: REAL);
         CIRCLE: (DIAMETER: REAL)
     END;

     FOO = RECORD
       CASE BOOLEAN OF
         TRUE: (I, J: INTEGER);
         FALSE: (CASE COLOR OF
                   BLUE: (X: REAL);
                   RED:  (Y: INTEGER4));
     END;
```

Only one variant part per record is allowed; it must be the last field of the record. However, this variant part can also have a variant (and so on, to any level). All field identifiers in a given record type must be unique, even in different variants. For example, after declaring the record types above, you could create and then assign to the variables shown below:

```
VAR O, P : OBJECT;
    F, G : FOO;

BEGIN
  O.DIAMETER := 12.34;  {CASE of CIRCLE}
  P.SIZE := 1.2;        {CASE of SQUARE}
  F.I := 1; F.J := 2;   {CASE of TRUE}
  G.X := 123.45;        {CASE of FALSE and BLUE}
  G.Y := 678999         {CASE of FALSE and RED;}
                        {this overwrites G.X.}
END;
```


PRELIMINARY DRAFT

The latest ISO standard requires every possible tag field value to select some variant. Therefore, it is illegal to include CASE INTEGER OF and omit a variant for every possible INTEGER value. However, such an omission error is not caught in MS-Pascal.

MS-Pascal supports the use of full CASE constant options in the variant clause; that is, a list of constants can define a case. At the extend level, subranges and the OTHERWISE statement can also define a case. If used, OTHERWISE applies to the last variant in the list and is not followed by a colon. You can also declare an empty variant, such as POINT:() or OTHERWISE (). You can even declare an entirely empty record type, although the compiler issues a warning whenever the record is used.

The ISO standard defines a number of errors that relate to variant records; these errors may not be caught in MS-Pascal, even if the tag-checking switch is on. (The tag-checking switch generates code each time a variant field is used, to check that the tag value is correct.) In the record type declaration of OBJECT (in the previous example), any use of SIZE generates a check that S = SQUARE. However, in the case of FOO, uses of I cannot be checked because MS-Pascal does not allocate the BOOLEAN tagfield.

The ISO standard further declares that when a "change of variant" occurs (such as when a new tag value is assigned), all the variant fields become undefined. However, MS-Pascal does not set the fields to an uninitialized value when a new tag is assigned. Therefore, using a variant field with an undefined value is an error not caught in MS-Pascal.

MS-Pascal does not enforce various restrictions on a record variable allocated on the heap with the long form of the NEW procedure (see Chapter 14 for details). However, MS-Pascal does check an assignment to such a "short record" to see that only

PRELIMINARY DRAFT

the short record itself is modified in the heap.

A record allocated with the long form of NEW can be released using the short form of DISPOSE with no ill effects (this is an ISO error not caught in MS-Pascal). It is also an error not caught in MS-Pascal to DISPOSE of a record passed as a reference parameter or used by an active WITH statement.

Variant records interact with MS-Pascal features in two ways:

1. Declaring a variant that contains a file is not safe; any change to the file's data using a field in another variant may lead to I/O errors, even if the file is closed. In the following example, any use of R leads to errors in F:

```
RECORD CASE BOOLEAN OF
    TRUE : (F: FILE OF REAL);
    FALSE : (R:ARRAY [1..100] OF REAL);
    END;
```

2. Giving initial data to several overlapping variants in a variable in a VALUE section may have unpredictable results. In the following example, the initial value of LAP is uncertain:

```
VAR LAP : RECORD CASE BOOLEAN OF
    TRUE : (I: INTEGER4);
    FALSE : (R: REAL);
    END;
VALUE LAP.I := 10; LAP.R := 1.5;
```

MS-Pascal generates a warning message if you attempt either of these operations.

6.3.2 EXPLICIT FIELD OFFSETS

MS-Pascal lets you assign explicit byte offsets to

PRELIMINARY DRAFT

the fields in a record. This system level feature can be useful for interfacing to software in other languages, since other control block formats may not conform to the usual MS-Pascal field allocation method. Assigning explicit field offsets permits unsafe operations (such as overlapping fields and word values at odd byte boundaries), and is not recommended unless the interface is necessary.

Example showing assignment of explicit byte offsets:

```
TYPE CPM = RECORD
             NDRIVE [Ø0]: BYTE;
             FILENM [Ø1]: STRING (8);
             FILEXT [Ø9]: STRING (3);
             EXTENT [12]: BYTE;
             CPMRES [13]: STRING (2Ø);
             RECNUM [33]: WORD;
             RECOVF [35]: BYTE;
           END;

      OVERLAP = RECORD
             BYTEAR [Ø0]: ARRAY [Ø..7] OF BYTE;
             WORDAR [Ø0]: ARRAY [Ø..3] OF WORD;
             BITSAR [Ø0]: SET OF Ø..63;
           END;
```

As can be seen in the example, the offset is enclosed in brackets (similar to attribute notation). The number is the byte offset to the start of the field. Some target machines may not permit accessing a 16-bit value at an odd address, but the compiler doesn't catch this as an error.

If you give any field an offset, give offsets to all fields. For any offset that you omit, the compiler picks an arbitrary value. Although the compiler will process a declaration that includes both offsets and variant fields, you should use only one or the other in a given program.

Although you can completely control field overlap

PRELIMINARY DRAFT

with explicit offsets, variants provide the long forms of the procedures NEW, DISPOSE, and SIZEOF. If you want to allocate different length records, use the RETYPE and GETHQQ procedures, instead of variants and the long form of NEW. For example:

```
CPMPV := RETYPE (CPMP, GETHQQ (36));
```

The compiler does support structured constants for record types with explicit offsets. Internally, odd length fields greater than one are rounded to the next even length. For example:

```
ODDR = RECORD
         F1[00] : STRING (3);
         F2[03] : CHAR
       END;
```

In this example, field F1 is four bytes long, so an assignment to F1 overwrites F2. In such a record, all odd length fields must be assigned first.

6.4 SETS

A set type defines the range of values that a set may assume. This range of assumable values is the "power set" of the base type you specify in the type definition. The power set is the set of all possible sets that can be composed from an ordinal base type. The null set, [], is a member of every set.

Suppose you declare the following set types:

```
TYPE HUES = SET OF COLOR;
     CAPS = SET OF 'A'..'Z';
     MATTER = SET OF (ANIMAL, VEGETABLE, MINERAL);
```

Then you declare variables like the following:

```
VAR FLAG : HUES;
```


PRELIMINARY DRAFT

```
VOWELS : CAPS;
LIVE : MATTER;
```

Finally, you can assign these set variables:

```
FLAG := [RED, WHITE, BLUE];
VOWELS := ['A', 'E', 'I', 'O', 'U'];
LIVE := [ANIMAL, VEGETABLE];
```

The set elements must be enclosed in brackets. This practice differs from the use of parentheses to enclose the base enumerated type in a set type declaration.

Set operations are implemented directly by generated in-line code or by routines in the set unit. See Chapter 11 for a complete discussion of operations on sets.

The ORD value of the base type can range from Ø to 255. Thus, SET OF CHAR is legal, but SET OF 1942..1984 is not.

Sets whose maximum ORD value is 15 (sets that fit into a WORD) are usually more efficient than larger ones. Also, if the range checking switch is on, passing a set as a value parameter invokes a run-time compatibility check, unless the formal and actual sets have the same type.

Sets provide a clear and efficient way of giving several qualities or attributes to an object. In another language, you might assign each quality a power of two:

```
READY = 1
GETSET = 2
ACTIVE = 4
DONE = 8
```

You might then assign the qualities with a statement like this:

PRELIMINARY DRAFT

```
X := READY + ACTIVE)
```

and then test them using OR and AND as bitwise operators with a statement like:

```
IF ((X AND ACTIVE) <> Ø) THEN WRITELN ('GO FISH')
```

The equivalent declaration in MS-Pascal might be:

```
QUALITIES = SET OF (READY, GETSET, ACTIVE, DONE);
```

You could then assign the qualities with X := [GETSET, ACTIVE] and test them with the following operations:

IN	**tests a bit**
+	**sets a bit**
-	**clears a bit**

For example, an appropriate construction might be:

```
IF ACTIVE IN X THEN WRITELN ('GO FISH')
```

You can also use SET OF Ø..15 to test and set the bits in a WORD. Using WORDs both as a set of bits and as the WORD type requires giving two types to the word, with a variant record, the RETYPE function, or an address type.

The bits in a set are assigned starting with the most significant bit in the lowest addressed byte. Thus, on a byte-swapped machine, the set [Ø, 7, 8, 15] has the WORD value #8Ø + #Ø1 + #8ØØØ +#Ø1ØØ. See the MS-Pascal User's Guide for further details.

PRELIMINARY DRAFT

7. FILES

A file is a structure that consists of a sequence of components, all of the same type. MS-Pascal interfaces with a given operating system through files. Therefore, you must understand the FILE type in order to perform input to and output from a program.

7.1 DECLARING FILES

As with any other type, you must declare a file variable in order to use it. However, declaring a FILE type does not fix the number of components in a file.

Examples of FILE declarations:

```
TYPE F1 = FILE OF COLOR;
     F2 = FILE OF CHAR;
     F3 = TEXT;
```

Conceptually, a file is simply another data type, like an array, but with no bounds and with only one component accessible at a time. A file usually corresponds to one of the following:

1. Disk files

2. Terminals

3. Printers

4. Other input and output devices

This implies the following restriction in Pascal: a FILE OF FILE is illegal, directly or indirectly. Other structures, such as a FILE OF ARRAYs or an ARRAY OF FILEs, are permitted.

PRELIMINARY DRAFT

Most Pascal implementations connect file variables to the data files of the operating system. MS-Pascal always uses the target operating system to access files but does not impose additional formatting or structure on operating system files.

MS-Pascal supports normal statically allocated files, files as local variables (allocated on the stack), and files as pointer referents (allocated on the heap). Except for files in super arrays, the compiler generates code to initialize a file when it is allocated and to CLOSE a file when it is deallocated.

This initialization call occurs automatically in most cases. However, a file declared in a module or uninitialized unit's interface gets its initialization call only if you call the module or unit identifier as a procedure. File declarations in such cases get the following compiler warning:

Contains file initialize module

Only a file in an interface of an uninitialized unit does not generate this warning.

MS-Pascal sets up the standard files, INPUT and OUTPUT (discussed in Section 7.5). In standard Pascal, files must be given in the program header, and when you run your program, the run-time system prompts you for filenames. At the extend level, you may use the ASSIGN and READFN procedures to give explicit operating system filenames to files not included in the program header.

Files in record variants or super array types are not recommended; if you use them, the compiler issues a warning. A file variable cannot be assigned, compared, or passed by value: it can only be declared and passed as a reference parameter.

At the extend level, you can indicate a file's

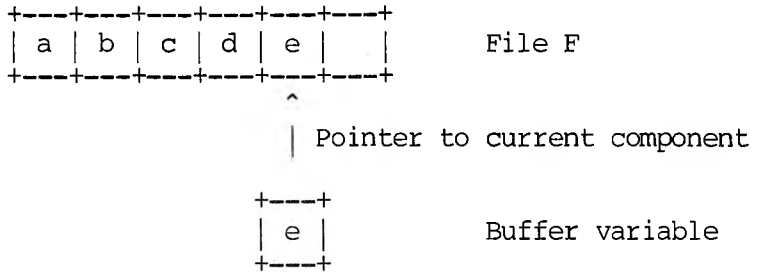
PRELIMINARY DRAFT

access method or other characteristics by specifying the mode of the file. The mode is a value of the predeclared enumerated type FILEMODES. The modes available normally include the three base modes, SEQUENTIAL, TERMINAL, and DIRECT. All files, except INPUT and OUTPUT, are given SEQUENTIAL mode by default. INPUT and OUTPUT are given the default mode TERMINAL.

7.2 THE BUFFER VARIABLE

Every file F has an associated buffer variable F^. A buffer variable and its associated file might look like Figure 7-1.

Figure 7-1: Buffer Variable and File



The procedures GET and PUT use this buffer variable to READ from and WRITE to files. GET copies the current component of the file to the buffer variable. PUT does the opposite; that is, PUT copies the value of the buffer variable to the current component.

The buffer variable can be referenced (its value fetched or stored) like any other MS-Pascal variable. This allows execution of assignments like the following:

PRELIMINARY DRAFT

```
F^ := 'z'
C := F^
```

A file buffer variable can be passed as a reference parameter to a procedure or function or used as a record in a WITH statement. However, the file buffer variable may not be updated correctly if the file position changes within the procedure, function, or WITH statement. The compiler issues a warning message to alert you to this possibility.

For example, the following use of a file buffer variable generates a warning at compile time:

```
VAR  A : TEXT;
PROCEDURE CHAR_PROC (VAR X : CHAR);
  .
  .
CHARPROC (A^);
{Warning issued here}
```

Two special internal mechanisms in MS-Pascal, lazy evaluation and concurrent I/O, allow, respectively, interactive terminal input in a natural way and overlapped I/O along with program execution. Lazy evaluation is applied to all ASCII structured files and is necessary for natural terminal input. Concurrent I/O is applied to all BINARY structured files and is necessary for some operating systems that support overlapping input and output.

Both mechanisms generate a run-time call that is executed before any use of the buffer variable. See Sections 15.1.5 and 15.1.6 for complete details.

7.3 FILE STRUCTURES

MS-Pascal files have two basic structures: BINARY and ASCII. These two structures correspond to raw data files and human-readable textfiles, respectively.

PRELIMINARY DRAFT

7.3.1 BINARY STRUCTURE FILES

The Pascal data type FILE OF <type> corresponds to MS-Pascal BINARY structure files. These, in turn, correspond to unformatted operating system files.

Under operating systems that divide files into records, every record is one component of the file type (not to be confused with the record type). Primitive procedures such as GET and PUT operate on a record basis. Under operating systems that do not have their own record structure, the primitive procedures GET and PUT transfer a fixed number of bytes per call, equal to the length of one component. See Section 7.4 for further discussion of BINARY files.

7.3.2 ASCII STRUCTURE FILES

The Pascal data type TEXT corresponds to MS-Pascal ASCII structure files. These, in turn, correspond to textual operating system files (called "textfiles" in this manual).

The Pascal TEXT type is like a FILE OF CHAR, except that groups of characters are organized into "lines" and, to a lesser extent, "pages." Primitive file procedures, such as GET and PUT, always operate on a character basis.

However, under operating systems that divide files into records, every record is a line (not a character). Even in operating systems that do not have their own record structure, other languages and utilities have some way of organizing bytes into lines of characters.

MS-Pascal provides a number of special functions and procedures that use this line-division feature.

PRELIMINARY DRAFT

Because MS-Pascal does not impose any additional formatting on operating system files of most modes (including SEQUENTIAL, TERMINAL, and DIRECT), programs in other languages can generate and use these files.

Pascal textfiles (files of type TEXT) are divided into lines with a "line marker," conceptually a character not of the type CHAR. In theory, a textfile can contain any value of type CHAR. However, under some operating systems, writing a particular character (say, CHR (13), carriage return, or CHR (1Ø), line feed) terminates the current line (record). This character value is the line marker in this case and, when read, always looks like a blank.

Under other operating systems, there may be no a terminating character. Still, as far as you are concerned, every line is followed by a line marker that reads as a blank.

At the extend level, a declaration for a textfile may include an optional line length. Setting the line length, which sets record length, is needed only for DIRECT mode textfiles. You can specify line length for other modes as well, but doing so has no effect.

Specify the line length of a textfile as a constant in parentheses after the word TEXT:

```
TYPE NAMEADDR = TEXT (128);
     DEFAULTX = TEXT;
     SMALLBUF = TEXT (2);
```

7.4 FILE ACCESS MODES

The file modes in MS-Pascal are SEQUENTIAL, TERMINAL, and DIRECT. SEQUENTIAL and TERMINAL mode files are available at the standard level; all

PRELIMINARY DRAFT

three, including DIRECT mode, are available at the extend level. SEQUENTIAL and TERMINAL mode ASCII structure files can have variable length records (lines); DIRECT mode files must have fixed length records or lines.

The declaration of a file in Pascal implies its structure, but not its mode. For example, FILE OF STRING (80) indicates BINARY structure; TEXT indicates ASCII structure. An assignment like F.MODE := DIRECT sets the mode; this works only at the extend level and is currently needed only to set DIRECT mode.

7.4.1 TERMINAL MODE FILES

TERMINAL mode files always correspond to an interactive terminal or printer. TERMINAL mode files, like SEQUENTIAL mode files, are opened at the beginning of the file for either reading or writing. Records are accessed one after the other until the end of the file.

Operation of TERMINAL mode input for terminals depends on the file structure (ASCII or BINARY). For ASCII structure (type TEXT), entire lines are read at one time. TEXT type permits the usual operating system intraline editing, including backspace, advance cursor, and cancel. Characters are echoed to the terminal screen while the line is typed.

If the target operating system does not support intraline editing or echo, the MS-Pascal file system interface provides it. However, since an entire line is read at once, you cannot read the characters as you type them, cannot invoke several prompts and responses on the same line, and so on.

For BINARY structure TERMINAL mode (usually type FILE OF CHAR), you can read characters as you type

PRELIMINARY DRAFT

them. No intraline editing or echoing is done. This method permits screen editing, menu selection, and other interactive programming on a keystroke rather than line basis.

TERMINAL mode files use lazy evaluation to properly handle normal interactive reading of the terminal keyboard. See Section 15.1.5 for details.

7.4.2 SEQUENTIAL MODE FILES

SEQUENTIAL mode files are generally disk files or other sequential access devices. Like TERMINAL mode files, SEQUENTIAL mode files are opened at the beginning of the file for either reading or writing, and records are accessed one after another until the end of the file. Standard Pascal files are in SEQUENTIAL mode by default (except for INPUT and OUTPUT).

7.4.3 DIRECT MODE FILES

DIRECT mode files are generally disk files or other random access devices. DIRECT mode files and the ability to access the mode of a file are available at the extend level of MS-Pascal.

DIRECT mode ASCII structure files, as well as all BINARY structure files, have fixed-length records, where a record is either a line or file component. (Here the term "record" refers not to the normal Pascal record type, but to a disk structuring unit.) DIRECT files are always opened for both reading and writing, and records can be accessed randomly by record number. There is no record number zero; records begin with record number one.

PRELIMINARY DRAFT

7.5 THE PREDECLARED FILES INPUT AND OUTPUT

Two files, INPUT and OUTPUT, are predeclared in every MS-Pascal program. These files get special treatment as program parameters and are normally required as parameters in the program heading:

```
PROGRAM ACTION (INPUT, OUTPUT);
```

If there are no program parameters and the program does not use the files INPUT and OUTPUT, the heading can look like this:

```
PROGRAM ACTION;
```

However, you should include INPUT and OUTPUT as program parameters if you use them, either explicitly or implicitly, in the program itself:

```
WRITE (OUTPUT, 'Prompt: ')        {explicit use}
WRITE ('Prompt: ')                {implicit use}
```

These examples generate a warning if OUTPUT is not declared in the program heading. The only effect of INPUT and OUTPUT as program parameters is to suppress this warning.

Although you can redefine the identifiers INPUT and OUTPUT, the file assumed by textfile input and output procedures and functions (e.g., READ, EOLN) is the predeclared definition. The procedures RESET (INPUT) and REWRITE (OUTPUT) are generated automatically, whether or not INPUT and OUTPUT are present as program parameters (you may also use these procedures explicitly).

INPUT and OUTPUT have ASCII structure and TERMINAL mode. They are initially connected to your terminal and opened automatically. At the extend level of MS-Pascal, you can change these characteristics if you wish.

PRELIMINARY DRAFT

7.6 EXTEND LEVEL I/O

A file variable in MS-Pascal is really a record, of type FCBFQQ, called a file control block. At the extend level, a few standard fields in this record help you handle file modes and error trapping.

Additional fields and the record type FCBFQQ itself can be used at the system level, described in Section 7.7. Along with access to certain FCB fields, extend level I/O also includes the following procedures:

ASSIGN	READFN
CLOSE	READSET
DISCARD	SEEK

See Section 15.3 for a description of these procedures.

Use the normal record field syntax to access FCB fields. For a file F, the fields are named F.MODE, F.TRAP, and F.ERRS. You can change or examine these fields at any time. The following paragraphs describe the FCB fields.

F.MODE: FILEMODES

This field contains the mode of the file: SEQUENTIAL, TERMINAL, or DIRECT. These values are constants of the predeclared enumerated type FILEMODES. The file system uses the MODE field only during RESET and REWRITE. Thus, changing the MODE field of an open file has no effect and is, in fact, discouraged. Except for INPUT and OUTPUT, which have TERMINAL mode, a file's mode is SEQUENTIAL by default.

RESET and REWRITE change the mode from SEQUENTIAL to TERMINAL if the device being opened is a terminal or printer and if the target operating system allows

PRELIMINARY DRAFT

it. This change is useful in programs designed to work either interactively or in batch mode. You must set DIRECT mode before RESET or REWRITE if you plan to use SEEK on a file.

F.TRAP: BOOLEAN

If this field is TRUE, error trapping for file F is turned on. Then, if an input/output error occurs, the program does not abort and the error code can be examined. Initially, F.TRAP is set FALSE. If FALSE and an I/O error occurs, the program aborts.

F.ERRS: WRD(Ø)..15

This field contains the error code for file F. An error code of zero means no error; values from 1 to 15 imply an error condition. If you attempt a file operation other than CLOSE or DISCARD and F.ERRS is not zero, the program immediately aborts if F.TRAP is FALSE. However, if F.TRAP is TRUE, the attempted file operation is ignored, and the program continues.

CLOSE and DISCARD do not examine the initial value of F.ERRS, so they are never ignored and do not cause an immediate abort. Nevertheless, if CLOSE or DISCARD themselves generate an error condition, F.TRAP determines whether to trap the error or to abort.

An operation ignored because of an error condition does not change the file itself, but may change the buffer variable or READ procedure input variables. See Appendix H for a complete listing of error messages and warnings.

Also at the extend level, you can set the line length for a textfile, as shown:

```
TYPE SMALLBUF = TEXT (16);
VAR RANDOMTEXT: TEXT (132);
```


PRELIMINARY DRAFT

Declaring line length applies only to DIRECT mode ASCII structure files, where the line length is the record length used for reading and writing. Setting the line length has no effect on other ASCII files.

7.7 SYSTEM LEVEL I/O

At the system level of MS-Pascal, you can call procedures and functions that have a formal reference parameter of type FCBFQQ with an actual parameter of the type FILE OF <type> or TEXT, or the identical FCBFQQ type.

The FCBFQQ type is the underlying record type used to implement the file type in MS-Pascal. The interface for the target system FCBFQQ type (and any other types needed) is usually part of the internal file system. Thus, procedures and functions that reference FCBFQQ parameters can be called with any file type, including predeclared procedures and functions like CLOSE and READ.

An FCBFQQ type variable can be passed to procedures like READLN and WRITELN that require a textfile. This permits, for example, calling directly the interface routines on the target operating system, working with mixtures of MS-Pascal and MS-FORTRAN (which share the file system interface but have special FCBFQQ fields), and other special file system activities.

Such activities require a sound knowledge of the file system. See Section 8.2 in the MS-Pascal User's Guide for a discussion of the file system interface and file control block.

PRELIMINARY DRAFT

8. REFERENCE AND OTHER TYPES

The array, record, and set types discussed in Chapter 6 let you describe data structures whose form and size are predetermined and whose components are accessed in a standard way. The file type, described in Chapter 7, is a structure that varies in size but whose form and means of access are predetermined.

This chapter discusses reference types, which allow data structures that vary in size and form and whose means of access is particular to the programming problem involved. Also included are notes on PACKED types and procedural and functional types.

8.1 REFERENCE TYPES

A reference to a variable or constant is an indirect way to access it. The pointer type is an abstract type for creating, using, and destroying variables allocated from an area called the heap. The heap is a dynamically growing and shrinking region of memory allocated for pointer variables.

MS-Pascal also provides two machine-oriented address types: one for addresses that can be represented in 16 bits, the other for addresses that require 32 bits.

Pointers are generally used for trees, graphs, and list processing. Use of pointers is portable, structured, and relatively safe.

Address types provide an interface to the hardware and operating system; their use is frequently unstructured, machine specific, low level, and unsafe. Both pointers and address types are discussed further in the following sections.

PRELIMINARY DRAFT

8.1.1 POINTER TYPES

A pointer type is a set of values that point to variables of a given type. The type of the variables pointed to is called the "reference type." Reference variables are all dynamically allocated from the heap with the NEW procedure. Pascal variables are normally allocated on the stack or at fixed locations.

You can perform only the following actions on pointers:

1. Assign them

2. Test them for equality and inequality with the two operators = and <>

3. Pass them as value or reference parameters

4. Dereference them with the up arrow (^)

Every pointer type includes the pointer value NIL. Pointers are frequently used to create list structures of records, as shown in the following example:

```
TYPE
  TREETIP = ^ TREE;
  TREE = RECORD
           VAL: INTEGER;
              {Value of TREE cell.}
           LEFT, RIGHT:  TREETIP
              {Pointers to other TREETIP cells.}
              {Note recursive definition.}
         END;
```

Unlike most type declarations, a pointer type declaration can refer to a type of which it is itself a component. The declaration can also refer to a type declared later in the same TYPE section,

PRELIMINARY DRAFT

as in TREE and TREETIP in the previous example. Such a declaration is called a forward pointer declaration and permits recursive and mutually recursive structures. Because pointers are often used in list structures, forward pointer declarations occur frequently.

The compiler checks for one ambiguous pointer declaration. Suppose the previous example is in a procedure nested in another procedure that also declared a type TREE. Then the reference type of TREETIP could be either the outer definition or the one following in the same TYPE section. MS-Pascal assumes the TREE type intended is the one later in the same TYPE section and gives the warning:

Pointer Type Assumed Forward

At the extend level, a pointer can have a super array type as a referent type. The actual upper bounds of the array are passed to the NEW procedure to create a heap variable of the correct size. Forward pointer declarations of the super array type are not allowed.

The ISO standard requires strict compatibility between pointers. For example, you cannot declare two pointers with different types and then assign or compare them, even if they happen to point to the same underlying type. For example:

```
VAR PRA : ^ REAL;
    PRE : ^ REAL;
BEGIN PRA := PRE END;  {This is illegal!}
```

Programs usually contain only one type declaration for a pointer to a given type. In the TREETIP example, the type of LEFT and RIGHT could be ^TREE instead of TREETIP, but then you couldn't assign variables of type TREETIP to these fields. However, it is sometimes useful to make sure that two classes of pointers are not used together, even if they

point to the same type.

For example, suppose you have a type RESOURCE kept in a list and declare two types, OWNER and USER, of type ^RESOURCE. The compiler would catch assignment of OWNER values to USER variables and vice versa and issue a warning message.

In theory, pointers have nothing to do with actual machine addresses. In fact, a pointer may be implemented in different ways on different target machines. A pointer can be implemented as a normal address, as a segment offset address, as an offset from one or more fixed locations, or as an indirect address, among other possibilities.

If the initialization checking switch is on, a newly created pointer has an uninitialized value. If the NIL checking switch is on, pointer values are tested for various invalid values. Invalid values include NIL, uninitialized values, reference to a heap item that has been DISPOSEd, or a value that is not valid as a heap reference.

8.1.2 ADDRESS TYPES

As a system implementation language, MS-Pascal needs a method of creating, manipulating, and dereferencing actual machine addresses. The pointer type is applicable only to variables in the heap.

There are two kinds of addresses: relative and segmented. The keywords ADR and ADS refer to the relative address type and the segmented address type, respectively. As the following example shows, you use the keywords both as type clause prefixes and as prefix operators:

```
VAR INT_VAR  :  INTEGER;
    REAL_VAR :  REAL;
    A_INT    :  ADR OF INTEGER;
```


PRELIMINARY DRAFT

```
    {Declaration of ADR variable}
    AS_REAL  :  ADS OF REAL;
    {Declaration of ADS variable}

BEGIN
  INT_VAR  :=  1;
  {Normal integer variable}
  REAL_VAR :=  3.1415;
  {Normal real variable}
  A_INT    :=  ADR INT_VAR;
  {ADR used as operator}
  AS_REAL  :=  ADS REAL_VAR;
  {ADS used as operator}
  WRITELN (A_INT^,AS_REAL^)
  {Note use of up arrow to dereference}
  {the address types.}
  {Output is 1 and 3.1415.}
END.
```

The characteristics of relative and segmented address types, as implemented for different machines, are shown in Table 8-1.

Table 8-1: Relative and Segmented Machine Addresses

MACHINE	ADR	ADS
8080	16-bit absolute	Same as ADR
8086	16-bit default data segment offset	16-bit offset, 16-bit segment
Z8000 (unsegmented)	16-bit data absolute	Same as ADR
Z8000 (segmented)	Same as ADS	16-bit segment, 16-bit offset

See your MS-Pascal User's Guide for details specific to your implementation of the compiler.

PRELIMINARY DRAFT

In MS-Pascal, you may declare a variable that is an address:

```
VAR X :  ADR OF BYTE;
```

Then, with the following record notation, you can assign numeric values to the actual variable:

```
X.R := 16#FFFF
```

In an unsegmented environment, the .R (relative address) is the only record field available for ADR and ADS addresses.

Since MS-Pascal allows nondecimal numbering, you may specify the assigned value in hexadecimal notation. You may also assign to a segment field with the ADS type in a segmented environment, using the field notation .S (segment address). Thus, you may declare a variable of an ADS type and then assign values to its two fields:

```
VAR Y : ADS OF WORD;
  .
  .
Y.S := 16#0001
Y.R := 16#FFFF
```

As shown above, any 16-bit value can be directly assigned to address type variables, using the .R and .S fields. The ADR and ADS operators obtain these addresses directly. The example below assigns addresses this way to the variables X and Y:

```
VAR  X : ADR OF BYTE;
     Y : ADS OF WORD;
     W : WORD;
     B : BYTE;
       .
       .
     X := ADR B;
     Y := ADS W;
```


PRELIMINARY DRAFT

MS-Pascal supports the following two predeclared address types:

```
ADRMEM = ADR OF ARRAY [Ø..32766] OF BYTE;
ADSMEM = ADS OF ARRAY [Ø..32766] OF BYTE;
```

Since the type referred to by the address is an array of bytes, byte indexing is possible. For example, if A is of type ADRMEM, then A^[15] is the byte at the address A.R + 15, where .R specifies an actual 16-bit address.

You can use the address types for a constant address (a form of structured constant); you can also take the address of a constant or expression. For example:

```
TYPE ADRWORD = ADR OF WORD;
     ADSWORD = ADS OF WORD;
VAR W: WORD;
    R: ADRWORD;
CONST CONADR = ADRWORD (1234);
BEGIN
  W := CONADR^;
  {Get word at address 1234}
  W := ADSWORD (Ø, 32)^;
  {Get word at address Ø:32}
  W := (ADS W).S;
  {Get value of DS segment register}
  R := ADR '123';
  {Get address of a constant value}
  R := ADR (W DIV 2 + 1);
  {Get address of expression value}
END;
```

However, constants or expressions that yield addresses cannot currently be used as the target of an assignment (or as a reference parameter or WITH record), as shown:

PRELIMINARY DRAFT

```
CONST ADSCON = ADSWORD (256, 64);    {OK}
FUNCTION SOME_ADDRESS: ADSWORD;      {OK}
BEGIN
  ADSWORD (0, 32)^ := W; {Not permitted}
  ADSCON^ := 12;         {Not permitted}
  SOME_ADDRESS^ := 100;  {Not permitted}
END;
```

8.1.3 SEGMENT PARAMETERS FOR THE ADDRESS TYPES

Two keywords, VARS and CONSTS, are available as parameter prefixes, like VAR and CONST, to pass the segmented address of a variable. If P is of type ADS FOO, then P^ can be passed to a VARS formal parameter, such as VARS X: FOO, but cannot be passed to a VAR formal parameter.

In a segmented machine environment, a default data segment is assumed, in which case a VAR parameter is passed as the default data segment offset of a variable. A VARS parameter is passed as both the segment value and the offset value.

In the 8086 environment, both VARS parameters and ADS variables have the offset (.R) value in the word with the lower address and the segment (.S) value in the address plus two.

In the segmented Z8000 environment, the segment (.S) value is in the lower address and the offset (.R) value in the the address plus two. Also, the ADR type is identical to the ADS type.

In the nonsegmented environment (e.g., 8080), VAR and CONST are identical to VARS and CONSTS. Since ADS and ADR are identical in a nonsegmented environment, the ADS type is useful in situations where the target environment may change. For example, in MS-Pascal, some primitive file system calls are declared with ADS parameters.

PRELIMINARY DRAFT

In pointer type declarations, the up arrow (^) prefixes the type pointed to; in program statements, it dereferences a pointer so that the value pointed to can be assigned or operated on. The up arrow also dereferences ADR and ADS types in program statements.

Component selection with the up arrow (^) is performed before the unary operators ADR or ADS. Because the up arrow (^) selector can appear after any address variable to produce a new variable, it can occur, for example, in the target of an assignment, a reference parameter, as well as in expressions. Since ADS and ADR are prefix operators, they are used only in expressions, where they apply only to a variable or constant or expression.

Pascal is a strongly typed language; two pointer variables are compatible only if they have the same type (it is not enough that they point to the same type). However, two address types are considered the same type if they are both ADR or both ADS types. You can, for example, assign an ADR OF WORD to an ADR OF STRING (200). Such an assignment makes it easy to wipe out part of memory by assigning a variable of type STRING (200) to the 200 bytes starting at the address of a WORD variable.

If P1 is type ADR OF STRING (200) and P2 is any ADR OF type, the assignment P1^ := P2^ generates fast code with no range checking. Although this capability is not safe, operating systems and other software sometimes require it.

ADR and ADS are not compatible with each other, but the .R notation should overcome or reduce the problem.

8.1.4 USING THE ADDRESS TYPES

PRELIMINARY DRAFT

Within limits, you can combine and intermingle the two address types. The following example illustrates the rules that apply in a segmented environment:

```
VAR
  P: ADS OF DATA;
  {P is segmented address of type DATA.}
  Q: ADR OF DATA;
  {Q is relative address of type DATA.}
  X: DATA;
  {X is some variable of type DATA.}

BEGIN
  P := ADS X;
  {Assign the address of X to P.}
  X := P^;
  {Assign to X the value pointed to by P.}
  P := ADS P^;
  {Assign to P the address of the value whose}
  {address is pointed to by P. P is unchanged}
  {by this assignment.}
  Q := ADR X;
  {Assign the relative address of X to Q.}
  Q.R  := (ADR X).R;
  {Assign the relative address of X to Q,}
  {using the WORD type.}
  P := ADS Q^;
  {Assign address of variable at Q to P.}
  {You can always apply ADS to ADR^.}
  Q := ADR P^;
  {Illegal; you cannot apply ADR to ADS ^.}
  P.R := 16#8000;
  {Assign 32768 to P's offset field.}
  P.S := 16;
  {Assign 16 to P's segment field.}
  Q.R := P.R + 4;
  {Assign P's offset plus 4 to be the value of Q.}
END;
```

See also the examples given in Section 8.1.2.

PRELIMINARY DRAFT

8.1.5 NOTES ON REFERENCE TYPES

The address type and pointer type should be treated as two distinct types. The pointer type, in theory, is just an undefined mapping from a variable to another variable. The method of implementation is undefined. However, the address type deals with actual machine addresses.

Therefore, the pointer type is an abstract data type that works the same in all implementations; the address type is generally not portable, unless used with some caution. Address types are portable only if you restrict yourself to using ADS and never assign to fields. Even with these restrictions, however, address types can be quite useful.

The following special facilities that use pointer variables are not allowed with address variables.

1. The NEW and DISPOSE procedures are permitted only with pointers. NIL does not apply to the address type. There are no special address values for empty, uninitialized, or invalid addresses.

2. The type "address of super array type" is not supported in the same way as "pointer to super array type." Getting the address of a super array variable is still permitted with ADR and ADS. For example, if a procedure or function formal parameter is declared as VAR S: STRING, then within the procedure or function, the expression ADS S is fine. Unlike a pointer, the address does not contain any upper bounds.

8.2 PACKED TYPES

Any of the structured types can be PACKED. You can use PACKED types to save storage at the possible

PRELIMINARY DRAFT

expense of access time or access code space. However, in MS-Pascal, some limitations on the use of PACKED structures apply:

1. The prefix PACKED is always ignored, except for type checking, in sets, files, and arrays of characters, and in most versions of MS-Pascal has no actual effect on the representation of records and other arrays. Furthermore, PACKED can precede only one of the structure names ARRAY, RECORD, SET, or FILE; it cannot precede a type identifier. For example, if COLORMAP is the identifier for an unpacked array type, "PACKED COLORMAP" is not accepted.

2. A component of a PACKED structure cannot be passed as a reference parameter or used as the record of a WITH statement, unless the structure is of a string type. Also, you cannot obtain the address of a PACKED component with ADR or ADS.

3. A PACKED prefix applies only to the structure being defined: any components of that structure that are also structures are not packed unless you explicitly include the reserved word PACKED in their definition. The only exception to this rule, n-dimensional arrays, is discussed in Section 6.1.

8.3 PROCEDURAL AND FUNCTIONAL TYPES

Procedural and functional types are different from other MS-Pascal types. (Wherever the term "procedural" is used from here on, both procedural and functional is implied.) You cannot declare an identifier for a procedural type in a TYPE section; nor can you declare a variable of a procedural type. However, you can use procedural types to declare the type of a procedural parameter. In this sense they conform to the Pascal idea of a type.

PRELIMINARY DRAFT

A procedural type defines a procedure or function heading and gives any parameters and for a function the result type. The syntax of a procedural type is the same as a procedure or function heading, including any attributes. There are no procedural variables in MS-Pascal, only procedural parameters.

Example of a procedural type declaration:

```
PROCEDURE ZEROPOINT (FUNCTION FUN (X, Y: REAL):REAL)
```

The parameter identifiers in a procedural type (X and Y in the previous example) are ignored; only their type is important.

See Section 13.4.3 for more information about procedural types in MS-Pascal.

PRELIMINARY DRAFT

PRELIMINARY DRAFT

9. CONSTANTS

9.1 WHAT IS A CONSTANT?

A constant is a value that is known before a program starts and that does not change as the program progresses. Examples of constants include the number of days in the week, your birthdate, the name of your dog, or the phases of the moon.

A constant can be given an identifier, but you cannot alter the value associated with that identifier during the execution of the program. When you declare a constant, its identifier becomes a synonym for the constant itself.

Each constant implicitly belongs to some category of data:

1. Numeric constants (discussed in Section 9.3) are one of these number types: REAL, INTEGER, WORD, or INTEGER4.

2. Character constants (discussed in Section 9.4) are strings of characters enclosed in single quotation marks and are called "string literals" in MS-Pascal.

3. Available at the extend level, structured constants (discussed in Section 9.5) include constant arrays, records, and typed sets.

Also available at the extend level, constant expressions (discussed in Section 9.6) let you compute a constant based on the values of previously declared constants in expressions.

The identifiers defined in an enumerated type are constants of that type and cannot be used directly with numeric (or string) constant expressions. These identifiers can be used with the ORD, WRD, or

PRELIMINARY DRAFT

CHR functions (e.g., ORD (BLUE)). The extend level also permits directly reading and writing the enumerated type's constant identifiers as character strings.

TRUE and FALSE are predeclared constants of type BOOLEAN and can be redeclared. NIL is a constant of any pointer type; however, because it is a reserved word, you cannot redefine it. Also, the null set is a constant of any set type.

Numeric statement labels have nothing to do with numeric constants; you may not use a constant identifier or expression as a label. Internally, all constants are limited in length to a maximum of 255 bytes.

9.2 DECLARING CONSTANT IDENTIFIERS

Declaring a constant identifier introduces the identifier as a synonym for the constant. You put these declarations in the CONST section of a compiland, procedure, or function.

The general form of a constant identifier declaration is the identifier followed by an equals sign and the constant value. The following program fragment includes three statements that identify constants (beginning after the word "CONST"):

```
PROGRAM DEMO (INPUT, OUTPUT);
CONST DAYSINYEAR = 365;
      DAYSINWEEK = 7;
      NAMEOFPLANET = 'EARTH';
```

In this example, the numbers 365 and 7 are numeric constants; 'EARTH' is a string literal constant and must be enclosed in single quotation marks.

When you compile a program, the constant identifiers are not actually defined until after the

PRELIMINARY DRAFT

declarations are processed. Thus, a constant declaration like the following has no meaning:

```
N = -N
```

The ISO standard defines a strict order in which to set out the declarations in the declaration section of a program:

```
CONST MAX = 10;
TYPE NAME = PACKED ARRAY [1..MAX] OF CHAR;
VAR FIRST : NAME;
```

MS-Pascal relaxes this order and, in fact, allows more than one instance of each kind of declaration:

```
TYPE  COMPLEX = RECORD R, I : REAL END;
CONST PII = COMPLEX (3.1416, 00);
VAR   PIX : COMPLEX;
TYPE  IVEC = ARRAY [1..3] OF COMPLEX;
CONST PIVEC = IVEC (PII, PII, COMPLEX (0.0, 1.0));
```

9.3 NUMERIC CONSTANTS

Numeric constants are irreducible numbers such as 45, 12.3, and 9E12. The notation of a numeric constant generally indicates its type: REAL, INTEGER, WORD, or INTEGER4.

Numbers can have a leading plus (+) or minus sign (-), except when the numbers are within expressions. Therefore:

```
ALPHA := +10      is legal.

ALPHA + -10       is illegal.
```

Blanks embedded within constants are not permitted.

The compiler truncates any number that exceeds a certain maximum number of characters and gives a

PRELIMINARY DRAFT

warning when this occurs. The maximum length of constants (either 19 or 31) is the same as the maximum length of identifiers. For the maximum length of constants and identifiers in a particular version of the language, see Appendix A in your MS-Pascal User's Guide.

The syntax for numeric constants applies not only to the actual text of programs, but also to the content of textfiles read by a program.

Examples of numeric constants:

```
123        +12.345     0.17    -26.0   26.0E12
-1.7E-10    17E+3     -17E3      1E1     007
```

Numeric constants can appear in any of the following:

1. CONST sections
2. Expressions
3. Type clauses
4. Set constants
5. Structured constants
6. CASE statement CASE constants
7. Variant record tag values

The different types of numeric constants are discussed in detail in the following sections.

9.3.1 REAL CONSTANTS

The type of a number is REAL if the number includes a decimal point or exponent. The REAL value range depends on the REAL number unit of the target

PRELIMINARY DRAFT

machine. Generally, either the IEEE or the Microsoft REAL number format is used. This provides about seven digits of precision, with a maximum value of about 1.7Ø141lE38.

There is, however, a distinction between REAL values and REAL constants. The REAL constant range may be a subset of the REAL value range. In MS-format, REAL numeric constants must be greater than or equal to 1.ØE-38 and less than 1.ØE+38. In IEEE format, REAL numeric constants are kept in double precision and can range from about 1E-3Ø6 to 1E3Ø6.

The compiler issues a warning if there is not at least one digit on each side of a decimal point. A REAL number starting or ending with a decimal point may be misleading. For example, because left parenthesis-period substitutes for left square bracket, and right parenthesis-period for right square bracket:

(.1+2.)

is interpreted as:

[1+2]

Scientific notation in REAL numbers (as in 1.23E-6 or 4E7) is supported. The decimal point and exponent sign are optional when an exponent is given. Both the uppercase "E" and the lowercase "e" are allowed in REAL numbers. "D" and "d" are also allowed to indicate an exponent. This flexibility gives MS-Pascal compatibility with other languages.

When IEEE REAL4 and REAL8 format are used, all real constants are stored in REAL8 (double precision) format. If you require a single precision REAL4 constant, declare a REAL4 variable and give it your real constant value in a VALUE section. (You may want to give this variable the READONLY attribute as well.)

PRELIMINARY DRAFT

Versions of the compiler that run on one machine but generate code for another may lose a small amount of significance in REAL constants.

9.3.2 INTEGER, WORD, AND INTEGER4 CONSTANTS

The type of a non-REAL numeric constant is INTEGER, WORD, or INTEGER4. Table 9-1 shows the range of values that constants of each of these types can assume.

Table 9-1: INTEGER, WORD, and INTEGER4 Constants

TYPE	RANGE OF VALUES (minimum/maximum)	PREDECLARED CONSTANT
INTEGER	-MAXINT to MAXINT	MAXINT=32767
WORD	Ø to MAXWORD	MAXWORD=65535
INTEGER4	-MAXINT4 to MAXINT4	MAXINT4=2147483647

MAXINT, MAXWORD, and MAXINT4 are all predeclared constant identifiers. One of three things happens when you declare a numeric constant identifier:

1. A constant identifier from -MAXINT to MAXINT becomes an INTEGER.

2. A constant identifier from MAXINT+1 to MAXWORD becomes a WORD.

3. A constant identifier from -MAXINT4 to -MAXINT-1 (or MAXWORD+1 to MAXINT4) becomes an INTEGER4.

However, any INTEGER type constant (including constant expressions and values from -32767 to -1) automatically changes to type WORD if necessary; if the INTEGER value is negative, 65536 is added to it

PRELIMINARY DRAFT

and the underlying 16-bit value is not changed.

For example, you can declare a subrange of type WORD as WRD(Ø)..127; the upper bound of 127 is automatically given the type WORD. The reverse, however, is not true; constants of type WORD are not automatically changed to type INTEGER.

The ORD and WRD functions also change the type of an ordinal constant to INTEGER or WORD. Also, any INTEGER or WORD constant automatically changes to type INTEGER4 if necessary, but the reverse is not true.

Examples of relevant conversions are given in Table 9-2.

Table 9-2: Constant Conversions

CONSTANT	ASSUMED TYPE
Ø	INTEGER could become WORD or INTEGER4
-32768	INTEGER4 only
32768	WORD could become INTEGER4
Ø..2ØØØØ	INTEGER subrange
Ø..5ØØØØ	WORD subrange
Ø..8ØØØØ	Invalid: no INTEGER4 subranges
-1..5ØØØØ	Invalid: becomes 65535..5ØØØØ (i.e., -1 is treated as 65536)

At the standard level, any numeric constant (literal or identifier) may have a plus (+) or minus (-) sign.

PRELIMINARY DRAFT

9.3.3 NONDECIMAL NUMBERING

At the extend level, MS-Pascal supports not only decimal number notation, but also numbers in hexadecimal, octal, binary, or other base numbering (the base can range from 2 to 36). The number sign (#) acts as a radix separator.

Examples of numbers in nondecimal notation:

```
16#FF02
10#987
 8#776
 2#111100
```

Leading zeros are recognized in the radix, so a number like 008#147 is permitted. In hexadecimal notation, upper or lowercase letters A through F are permitted. A nondecimal constant without the radix (such as #44) is assumed to be hexadecimal. Nondecimal notation does not imply a WORD constant and may be used for INTEGER, WORD, or INTEGER4 constants. You must not use nondecimal notation for REAL constants or numeric statement labels.

9.4 CHARACTER STRINGS

Most Pascal manuals refer to sequences of characters enclosed in single quotation marks as "strings." In MS-Pascal, they are called "string literals" to distinguish them from string constants, which may be expressions, or values of the STRING type.

A string constant contains from 1 to 255 characters. A string constant longer than one character is of type PACKED ARRAY [1..n] OF CHAR, also known in MS-Pascal as the type STRING (n). A string constant that contains just one character is of type CHAR. However, the type changes from CHAR to PACKED ARRAY [1..1] OF CHAR (e.g., STRING (1)) if necessary. For example, a constant ('A') of type CHAR could be

PRELIMINARY DRAFT

assigned to a variable of type STRING (1).

A literal apostrophe (single quotation mark) is represented by two adjacent single quotation marks (e.g., 'DON''T GO'). The null string ('') is not permitted. A string literal must fit on a line. The compiler recognizes string literals enclosed in double quotations marks (") or accent marks (`), instead of single quotation marks, but issues a warning message when it encounters them.

The constant expression feature (discussed in Section 9.6) permits string constants made up of concatenations of other string constants, including string constant identifiers, the CHR () function, and structured constants of type STRING. This feature is useful for representing string constants that are longer than a line or that contain nonprinting characters. For example:

```
'THIS IS UNDERLINED' * CHR(13) * STRING (DO 18 OF '_')
```

The LSTRING feature of MS-Pascal adds the super array type LSTRING. LSTRING is similar to PACKED ARRAY [Ø..n] OF CHAR, except that element Ø contains the length of the string, which can vary from Ø to a maximum of 255. (See Section 6.2.2 for a discussion of LSTRINGs.) For now, note that, if necessary, a constant of type STRING (n) or CHAR changes automatically to type LSTRING.

NULL is a predeclared constant for the null LSTRING, with the element Ø (the only element) equal to CHR (Ø). NULL cannot be concatenated, since it is not of type STRING. It is the only constant of type LSTRING.

Examples of string literal declarations:

```
NAME = 'John Jacob';      {a legal string literal}
LETTER = 'Z';             {LETTER is of type CHAR}
QUOTED_QUOTE = '''';      {Quotes quote}
```


PRELIMINARY DRAFT

```
NULL_STRING = NULL;        {legal}
NULL_STRING = '';          {illegal}
DOUBLE = "OK";             {generates a warning}
```

9.5 STRUCTURED CONSTANTS

Standard Pascal permits only the numeric and string constants already mentioned, the pointer constant value NIL, and untyped constant sets.

At the extend level of MS-Pascal, on the other hand, you can use constant arrays, records, and typed sets. Structured constants can be used anywhere a structured value is allowed, in expressions as well as in CONST and VALUE sections.

1. An array constant consists of a type identifier followed by a list of constant values in parentheses separated by commas.

   Example of an array constant:

```
TYPE   VECT_TYPE = ARRAY [-2..2] OF INTEGER;
CONST  VECT = VECT_TYPE (5, 4, 3, 2, 1);
VAR    A : VECT_TYPE;
VALUE  A := VECT;
```

2. A record constant consists of a type identifier followed by a list of constant values in parentheses separated by commas.

   Example of a record constant:

```
TYPE REC_TYPE = RECORD
       A, B: BYTE;
       C, D: CHAR;
     END;
CONST RECR = REC_TYPE (#20, 0, 'A', CHR (20));
VAR   FOO : REC_TYPE;
VALUE FOO := RECR;
```


PRELIMINARY DRAFT

3. A set constant consists of an optional set type identifier followed by set constant elements in square brackets. Set constant elements are separated by commas. A set constant element is either an ordinal constant, or two ordinal constants separated by two dots to indicate a range of constant values.

   Example of a set constant:

```
TYPE  COLOR_TYPE = SET OF
   (RED, BLUE, WHITE, GREY, GOLD);
CONST SETC = COLOR_TYPE [RED, WHITE .. GOLD];
VAR   RAINBOW : COLOR_TYPE;
VALUE RAINBOW := SETC;
```

A constant within a structured array or record constant must have a type that can be assigned to the corresponding component type. For records with variants, the value of a constant element corresponding to a tag field selects a variant, even if the tag field is empty. The number of constant elements must equal the number of components in the structure, except for super array type structured constants. Nested structured constants are permitted.

An array or record constant nested within another structured constant must still have the preceding type identifier. For this reason, a super array constant can have only one dimension (see Section 6.2). The size of the representation of a structured constant must be between 1 and 255 bytes. If this 255-byte limit is a problem, declare a structured variable with the READONLY attribute, and initialize its components in a VALUE section.

Example of a complex structured constant:

```
TYPE R3 = ARRAY [1..3] OF REAL;
TYPE SAMPLE = RECORD I: INTEGER;
                     A: R3;
```


PRELIMINARY DRAFT

```
                    CASE BOOLEAN OF
                    TRUE: (S: SET OF 'A'..'Z';
                           P: ^ SAMPLE);
                    FALSE:(X: INTEGER);
                END;
CONST SAMP_CONST= SAMPLE (27, R3 (1.4, 1.4, 1.4),
                TRUE, ['A','E','I'], NIL);
```

Constant elements can be repeated with the phrase DO <n> OF <constant>, so the previous example could include "DO 3 OF 1.4" instead of "1.4, 1.4, 1.4".

MS-Pascal does not support set constant expressions, such as ['_'] + LETTERS, or file constant expressions. The constant 'ABC' of type STRING (3) is equivalent to the structured constant STRING ('A', 'B', 'C'). LSTRING structured constants are not permitted; use the corresponding STRING constants instead.

Structured constants (and other structured values, such as variables and values returned from functions) can be passed by reference using CONST parameters. For more information, see Section 13.4.

There are two kinds of set constants: one with an explicit type, as in CHARSET ['A'..'Z'], and one with an unknown type, as in [20..40]. You can use either in an expression or to define the value of a constant identifier. Set constants with an explicit type may also be passed as a reference (CONST) parameter. Sets of unknown type are unpacked, but the type changes to PACKED if necessary. Passing sets by reference is generally more efficient than passing them as value parameters.

9.6 CONSTANT EXPRESSIONS

Constant expressions in MS-Pascal allow you to compute constants based on the values of previously declared constants in expressions. Constant expressions can also occur within program

PRELIMINARY DRAFT

statements.

Example of a constant expression declaration:

```
CONST HEIGHT_OF_LADDER = 6;
      HEIGHT_OF_MAN    = 6;
      REACH  = HEIGHT_OF_LADDER + HEIGHT_OF_MAN;
```

Because a constant expression can contain only constants that you have declared earlier, the following is illegal:

```
CONST MAX = A + B;
      A   = 10;
      B   = 20;
```

Certain functions can be used within constant expressions. For example:

```
CONST A = LOBYTE (-23) DIV 23;
      B = HIBYTE (-A);
```

Table 9-3 shows the functions and operators you can use with REAL, INTEGER, WORD, and other ordinal constants, such as enumerated and subrange constants.

Table 9.3. Constant Operators and Functions

TYPE OF OPERAND	FUNCTIONS AND OPERATORS
REAL, INTEGER	Unary plus (+) Unary minus (-)
INTEGER, WORD	+ DIV OR HIBYTE() - MOD NOT LOBYTE() * AND XOR BYWORD()
Ordinal types	< <= CHR() LOWER() > >= ORD() UPPER() = <> WRD()

PRELIMINARY DRAFT

Boolean	AND	NOT OR
ARRAY	LOWER()	UPPER()
Any type	SIZEOF()	RETYPE()

Examples of constant expressions:

```
CONST FOO = (100 + ORD('X')) * 8#100 + ORD('Y');
      MAXSIZE = 80;
      X = (MAXSIZE > 80) OR (IN_TYPE = PAPERTAPE);
      {X is a BOOLEAN constant}
```

In addition to the operators shown in Table 9-3 for numeric constants, you can use the string concatenation operator (*) with string constants, as follows:

```
CONST A = 'abcdef';
      M = CHR (109); {CHR is allowed}
      ATOM = A * 'ghijkl' * M;
      {ATOM = 'abcdefghijklm'}
```

These constants can span more than one line, but are still limited to the 255 character maximum. These string constant expressions are allowed wherever a string literal is allowed, except in metacommands.

PRELIMINARY DRAFT

10. VARIABLES AND VALUES

10.1 WHAT IS A VARIABLE?

A variable is a value that is expected to change during the course of a program. Every variable must be of a specific data type. A variable may have an identifier.

If A is a variable of type INTEGER, then the use of A in a program actually refers to the data denoted by A. For example:

```
VAR A: INTEGER;
   BEGIN
      A := 1;
      A := A + 1;
   END;
```

These statements first assign a value of 1 to the data denoted by A, and subsequently assign it a value of 2.

Variables are manipulated by using notation to denote the variable, in the simplest case, a variable identifier. In other cases--variables may be denoted by array indices or record fields or the dereferencing of pointer or address variables.

The compiler itself may create "hidden" variables, allocated on the stack, in circumstances like the following:

PRELIMINARY DRAFT

1. When you call a function that returns a structured result, the compiler allocates a variable in the caller for the result.

2. When you need the address of an expression (e.g., to pass it as a reference parameter or to use it as a WITH statement record or with ADR or ADS), the compiler allocates a variable for the value of the expression.

3. The initial and final values of a FOR loop may require allocating a variable.

4. When the compiler evaluates an expression, it may allocate a variable to store intermediate results.

5. Every WITH statement requires a variable to be allocated for the address of the WITH's record.

10.2 DECLARING A VARIABLE

A variable declaration consists of the identifier for the new variable, followed by a colon and a type. You may declare variables of the same type by giving a list of the variable identifiers, followed by their common type. For example:

```
VAR XCOORD, YCOORD: REAL
```

You can declare a variable in any of the following locations:

1. VAR section of a program, procedure, function, module, interface, or implementation.

2. Formal parameter list of a procedure, function, or procedural parameter.

PRELIMINARY DRAFT

In a VAR section, you can declare a variable to be of any legal type; in a formal parameter list, you can include only a type identifier. (You cannot declare a type in the heading of a procedure or function). For example:

```
ROCEDURE NAME (GEORGE: ARRAY [1..10] OF COLOR)
{Illegal; GEORGE is of a new type.}

VAR VECTOR_A: VECTOR (10)
{Legal; VECTOR (10) is a type derived from}
{a super type.}
```

Each declaration of a file variable F of type FILE OF T implies the declaration of a buffer variable of type T, denoted by F^. At the extend level, a file declaration also implies the declaration of a record variable of type FCBFQQ, whose fields are denoted as F.TRAP, F.ERRS, F.MODE, and so on. See Sections 7.2, and Section 7.6, for information on buffer variables and FCBFQQ fields, respectively.

10.3 THE VALUE SECTION

The VALUE section in MS-Pascal lets you give initial values to variables in a program, module, procedure, or function. You can also initialize the variable in an implementation, but not in an interface.

The VALUE section can include only statically allocated variables (any variable declared at the program, module, or implementation level, or a variable with the STATIC or PUBLIC attribute). Variables with the EXTERN or ORIGIN attribute cannot occur in a VALUE section, since they are not allocated by the compiler.

PRELIMINARY DRAFT

The VALUE section can contain assignments of constants to entire variables or to components of variables. For example:

```
AR  ALPHA : REAL;
    ID    : STRING (7);
    I     : INTEGER;

VALUE
    ALPHA := 2.23;
    ID[1] := 'J';
    I     := 1;
```

However, within a VALUE section, you cannot assign a variable to another variable. The last line in the following example is illegal, since I must be a constant:

```
CONST MAX = 10;
VAR I, J : INTEGER;
VALUE I := MAX;
      J := I;
```

If the $ROM metacommand is off, variables are initialized by loading the static data segment. If the $ROM metacommand is on, the VALUE section generates an error message since ROM-based systems usually cannot statically initialize data.

10.4 USING VARIABLES AND VALUES

At the standard level of MS-Pascal, denotation of a variable may designate one of three things:

1. An entire variable

2. A component of a variable

3. A variable referenced by a pointer

PRELIMINARY DRAFT

A value may be any of the following:

1. A variable

2. A constant

3. A function designator

4. A component of a value

5. A variable referenced by a reference value

At the extend level, a function can also return an array, record, or set. The same syntax used for variables can denote components of the structures these functions return.

This feature also allows you to dereference a reference type that is returned by a function. However, you can use the function designator only as a value, not as a variable. For example, the following is illegal:

```
F (X, Y)^ := 42;
```

Also at the extend level, you can declare constants of a structured type. Components of a structured constant use the same syntax as variables of the same type (see Section 9.6).

Examples of structured constant components:

```
TYPE REAL3 = ARRAY [1..3] OF REAL;
{an array type}
CONST PIES = REAL3 (3.14, 6.28, 9.42);
{an array constant}
  .
  .
  .
X := PIES [1] * PIES [3];
{i.e., 3.14 * 9.42}
Y := REAL3 (1.1, 2.2, 3.3) [2];
{i.e., 2.2}
```


PRELIMINARY DRAFT

PRELIMINARY DRAFT

10.4.1 COMPONENTS OF ENTIRE VARIABLES AND VALUES

At the standard level, a variable identifier denotes an entire variable. A variable, function designator, or constant denotes an entire value.

A component of a variable or value is denoted by the identifier followed by a selector that specifies the component. The form of a selector depends on the type of structure (array, record, file, or reference).

10.4.1.1 Indexed Variables And Values

A component of an array is denoted by the array variable or value, followed by an index expression. The index expression must be assignment compatible with the index type in the array type declaration. An index type must always be an ordinal type. The index itself must be enclosed in brackets following the array identifier.

Examples of indexed variables and values:

```
RRAY_OF_CHAR ['C']
{Denotes the Cth element.}

'STRING CONSTANT' [6]
{Denotes the 6th element, the letter 'G'.}

BETAMAX [12] [-3]
BETAMAX [12,-3]
{These two say the same thing.}

ARRAY_FUNCTION (A, B) [C, D]
{Denotes a component of a two-dimensional array}
{returned by ARRAY_FUNCTION (A, B). A and B are}
{actual parameters.}
```


PRELIMINARY DRAFT

You can specify the current length of an LSTRING variable, LSTR, in either of two ways:

1. With the notation LSTR [0], to access the length as a CHAR component

2. With the notation LSTR.LEN, to access the length as a BYTE value

10.4.1.2 Field Variables And Values

A component of a record is denoted by the record variable or value followed by the field identifier for the component. Fields are separated by a period or value only once. Within the WITH statement, you can use the field identifier of a record variable directly.

Examples of field variables and values:

```
ERSON.NAME := 'PETE'

PEOPLE.DRIVERS.NAME := 'JOAN'

WITH PEOPLE.DRIVERS DO NAME := 'GERI'

RECURSING_FUNC ('XYZ').BETA
{Selects BETA field of record returned}
{by the function named RECURSIVE_FUNC.}

COMPLEX_TYPE (1.2, 3.14).REAL_PART
```

Record field notation also applies to files for FCBFQQ fields, to address type values for numeric representations, and to LSTRINGs for the current length.

PRELIMINARY DRAFT

10.4.1.3 File Buffers And Fields

At any time, only one component of a file is accessible. The accessible component is determined by the current file position and represented by the buffer variable. Depending on the status of the buffer variable, fetching its value may first read the value from the file. (This is called "lazy evaluation"; see Section 15.1.5.)

If a file buffer variable is passed as a reference parameter or used as a record of a WITH statement, the compiler issues a warning to alert you to the fact that the value of the buffer variable may not be correct after the position of the file is changed with a GET or PUT procedure.

Examples of file reference variables:

```
NPUT^
ACCOUNTS_PAYABLE.FILE^
```

10.4.2 Reference Variables

Reference variables or values denote data that refers to some data type. There are three kinds of reference variables and values:

1. Pointer variables and values

2. ADR variables and values

3. ADS variables and values

In general, a reference variable or value "points" to a data object. Thus, the value of a reference variable or value is a reference to that data object. To obtain the actual data object pointed to, you must "dereference" the reference variable by appending an up arrow (^) to the variable or value.

PRELIMINARY DRAFT

Example using pointer values:

```
AR   P, Q : ^INTEGER;
{P and Q are pointers to integers.}

NEW (P); NEW (Q);
{P and Q are assigned reference values to}
{regions in memory corresponding to data}
{objects of type INTEGER.}

P := Q;
{P and Q now point to the same region
{in memory.}

P^ := 123;
{Assigns the value 123 to the INTEGER value}
{pointed to by P.  Since Q points to this}
{location as well, Q^ is also assigned 123.}
```

Using NIL^ is an error (since a NIL pointer does not reference anything). At the extend level, you can also append an up arrow (^) to a function designator for a function that returns a pointer or address type. In this case, the up arrow denotes the value referenced by the return value. This variable cannot be assigned to or passed as a reference parameter.

Examples of functions returning reference values:

```
ATA1 := FUNK1 (I, J)^
 {FUNK1 returns a reference value; the up arrow}
{dereferences the reference value returned,}
{assigning the referenced data to DATA1.}

DATA2 := FUNK2 (K, L)^.FOO [2]
{FUNK2 returns a reference value. The up arrow}
{dereferences the reference value returned.  In}
{this case, the dereferenced value is a record.}
{The array component FOO [2] of that record is}
{assigned to the variable DATA2.}
```


PRELIMINARY DRAFT

If P is of type ADR OF some type, then P.R denotes the address value of type WORD. If P is of type ADS OF some type, then P.R denotes the offset portion of the address and P.S denotes the segment portion of the address. Both portions are of type WOR

Examples of address variables:

```
UFF_ADR.R
DATA_AREA.S
```

10.5 ATTRIBUTES

At the extend level of MS-Pascal, a variable declaration or the heading of a procedure or function may include one or more attributes. A variable attribute gives special information about the variable to the compiler.

Table 10-1: displays the attributes provided by MS-Pascal for variables.

Table 10.1. Attributes for Variables

ATTRIBUTE	VARIABLE
STATIC	Allocated at a fixed location, not on the stack.
PUBLIC	Accessible by other modules with EXTERN, implies STATIC.
EXTERN	Declared PUBLIC in another module, implies STATIC.
ORIGIN	Located at specified address, implies STATIC.
PORT	I/O address, implies STATIC.
READONLY	Cannot be altered or written to.

PRELIMINARY DRAFT

The EXTERN attribute is also a procedure and function directive; PUBLIC and ORIGIN are also procedure and function attributes. See Section 13.3 for a discussion of procedure and function attributes and directives. Sections 10.5.1 through 10.5.5 discuss the variable attributes in detail.

You can give attributes for variables only in a VAR section. You cannot specifying variable attributes in a TYPE section or a procedure or function parameter list.

You give one or more attributes in the variable declaration, enclosed in brackets and separated by commas (if specifying more than one attribute).

The brackets may occur in either of two places:

1. An attribute in brackets after a variable identifier in a VAR section applies to that variable only.

2. An attribute in brackets after the reserved word VAR applies to all the variables in the section.

Examples that specify variable attributes:

```
VAR A, B, C [EXTERN] : INTEGER;
{Applies to C only.}

VAR [PUBLIC] A, B, C  : INTEGER;
{Applies to A, B, and C.}

VAR [PUBLIC] A, B, C [ORIGIN 16#1000] : INTEGER;
{A, B, and C are all PUBLIC. ORIGIN of C}
{is the absolute hexadecimal address 1000.}
```


PRELIMINARY DRAFT

10.5.1 THE STATIC ATTRIBUTE

The STATIC attribute gives a variable a unique, fixed location in memory. STATIC variables differ from procedure or function variables that are allocated on the stack or ones that are dynamically allocated on the heap. Use of STATIC variables can save time and code space, but increases data space.

All variables at the program, module, or unit level are automatically assigned a fixed memory location and given the STATIC attribute.

Functions and procedures that use STATIC variables can execute recursively, but STATIC variables must be used only for data common to all invocations. Since most of the other variable attributes imply the STATIC attribute, the trade-off between savings in time and code space or reduced data space apply to the PUBLIC, EXTERN, ORIGIN, and PORT attributes, as well.

Files declared in a procedure or function with the STATIC attribute are initialized when the routine is entered; they are closed when the routine terminates like other files. However, other STATIC variables are initialized only before program execution. This means that, except for open FILE variables, STATIC variables can be used to retain values between invocations of a procedure or function.

Examples of STATIC variable declarations:

```
VAR VECTOR [STATIC]: ARRAY [0..MAXVEC] OF INTEGER;
VAR [STATIC] I, J, K: 0..MAXVEC;
```

The STATIC attribute does not apply to procedures or functions, as some other attributes do.

PRELIMINARY DRAFT

10.5.2 THE PUBLIC AND EXTERN ATTRIBUTES

The PUBLIC attribute indicates a variable that can be accessed by other loaded modules; the EXTERN attribute identifies a variable that resides in some other loaded module. The identifier is passed to the target linker in the generated code object file (where it may be truncated if the linker imposes a length restriction).Variables given the PUBLIC or EXTERN attribute are implicitly STATIC.

Examples of PUBLIC and EXTERN variable declarations:

```
VAR [EXTERN] GLOBE1, GLOBE2: INTEGER;
{The variables GLOBE1 and GLOBE2 are declared}
{EXTERN, meaning that they must be declared}
{PUBLIC in some other loaded module.}

VAR BASE_PAGE [PUBLIC, ORIGIN #12FE]: BYTE;
{The variable BASE_PAGE is located at 12FE,}
{hexadecimal. Because it is also PUBLIC, it can}
{be accessed from other loaded modules that}
{declare BASE_PAGE with the EXTERN attribute.}
```

PUBLIC variables are usually allocated by the compiler, unless you also give them an ORIGIN. Giving a variable both the PUBLIC and ORIGIN attributes tells the loader that a global name has an absolute address. PUBLIC cannot be combined with PORT.

If both PUBLIC and ORIGIN are present, the compiler does not need the loader to resolve the address. However, the identifier is still passed to the linker for use by other modules.

EXTERN variables are not allocated by the compiler. Nor do they have an ORIGIN, since giving both EXTERN and ORIGIN implies two different ways to access the variable. The reserved word EXTERNAL is synonymous with EXTERN. This fact increases portability from

PRELIMINARY DRAFT

other Pascals, since others commonly use one of the two.

PRELIMINARY DRAFT

Variables in the interface of a unit are automatically given either the PUBLIC or EXTERN attribute. If a program, module, or unit USES an interface, its variables are made EXTERN; if you compile the IMPLEMENTATION of the interface, its variables are made PUBLIC.

10.5.3 THE ORIGIN AND PORT ATTRIBUTES

The ORIGIN attribute directs the compiler to locate a variable at a given memory address; the PORT attribute specifies some kind of I/O address. In either case, the address must be a constant of any ordinal type. I/O ports, interrupt vectors, operating system data, and other related data can be accessed with ORIGIN or PORT variables.

Examples of ORIGIN and STATIC variable declarations:

```
VAR KEYBOARDP [PORT 16#FFF2]: CHAR;
VAR INTRVECT [ORIGIN 8#200]: WORD;
```

Variables with ORIGIN or PORT attributes are implicitly STATIC.

These attributes also, inhibit common subexpression optimization. For example, if GATE has the ORIGIN attribute, the two statements X := GATE; Y := GATE access GATE twice in the order given, instead of using the first value for both assignments. This fact ensures correct operation if GATE is a memory-mapped input port. However, if GATE is passed as a reference parameter, references to the parameter may be optimized away. For this reason, PORT variables cannot be passed as reference parameters.

ORIGIN and PORT variables are never allocated or initialized by the compiler. The associated address indicates only where the variable is found. ORIGIN always implies a memory address, but the meaning of PORT varies with the implementation.

PRELIMINARY DRAFT

In most implementations, I/O is assumed to be memory mapped, so PORT is just a synonym for ORIGIN. Other implementations use the machine's native input and output instructions. Still others call port input and output routines for every access.

For more information on the PORT attribute, see Appendix A, of your MS-Pascal User's Guide.

Giving the PORT and ORIGIN attributes in brackets immediately following the VAR keyword is ambiguous and generates a error during compilation. (It would be unclear to the compiler whether all variables following should be at the same address or addresses should be assigned sequentially.)

```
VAR [ORIGIN 0] FIRST, SECOND: BYTE;
{ILLEGAL!}
```

ORIGIN (but not PORT) permits a segmented address using "segment: offset" notation.

```
VAR SEGVECT [ORIGIN 16#0001:16#FFFE]: WORD;
```

Currently, a variable with a segmented ORIGIN cannot be used as the control variable in a FOR statement.

10.5.4 THE READONLY ATTRIBUTE

The READONLY attribute prevents assignments to a variable. It also prevents the variable being passed as a VAR or VARS parameter. Also, a READONLY variable cannot be read with a READ statement or used as a FOR control variable. You may use READONLY with any of the other attributes.

PRELIMINARY DRAFT

Examples of READONLY variable declarations:

```
VAR INPORT [PORT 12, READONLY]: BYTE;
{INPORT is a READONLY PORT variable.}

VAR [READONLY] I, J [PUBLIC], K [EXTERN]: INTEGER;
{I, J, and K are all READONLY;}
{J is also PUBLIC; K is also EXTERN.}
```

CONST and CONSTS parameters, as well as FOR loop control variables (while in the body of the loop), are automatically given the READONLY attribute. READONLY is the only variable attribute that does not imply STATIC allocation.

A variable that is both READONLY and either PUBLIC or EXTERN in one source file is not necessarily READONLY when used in another source file. The READONLY attribute does not apply to procedures and functions.

10.5.5 COMBINING ATTRIBUTES

You can give a variable multiple attributes. Separate the attributes with commas and enclose the list in brackets, as shown:

```
VAR [STATIC]
 X, Y, Z [ORIGIN #FFFE, READONLY]: INTEGER;
```

In the preceding example, Z is a STATIC, READONLY variable with an ORIGIN at hexadecimal FFFE. These rules apply when you are combining attributes:

1. If you give a variable the EXTERN attribute, you cannot give it the PORT, ORIGIN, or PUBLIC attributes in the current compiland.

2. If you give a variable the PORT attribute, you cannot give it the ORIGIN, PUBLIC, or EXTERN attributes at all.

3. If you give a variable the ORIGIN attribute, you cannot also give it the PORT or EXTERN attributes. However, you can combine ORIGIN with PUBLIC.

4. If you give a variable the PUBLIC attribute, you cannot also give it the PORT or EXTERN attributes. However, you can combine PUBLIC with ORIGIN.

5. You can use STATIC and READONLY with any other attributes.