PASCAL-86 USER'S GUIDE

Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051

Order Number: 121539-005

PASCAL-86 USER'S GUIDE

Order Number: 121539-005

Intel Corporation, 3065 Bowers Avenue, Santa Clara, CA 95051

Additional copies of this manual or other Intel literature may be obtained from:

Literature Department
Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051

Intel retains the right to make changes to these specifications at any time, without notice. Contact your local sales office to obtain the latest specifications before placing your order.

Intel Corporation makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel Corporation assumes no responsibility for any errors that may appear in this document. Intel Corporation makes no commitment to update nor to keep current the information contained in this document.

Intel Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in an Intel product. No other circuit patent licenses are implied.

Intel software products are copyrighted by and shall remain the property of Intel Corporation. Use, duplication or disclosure is subject to restrictions stated in Intel's software license, or as defined in ASPR 7-104.9(a)(9).

No part of this document may be copied or reproduced in any form or by any means without the prior written consent of Intel Corporation.

The following are trademarks of Intel Corporation and its affiliates and may only be used to identify Intel products:

BITBUS
COMMputer
CREDIT
Data Pipeline
Genius
i
î
I^{2}ICE
ICE
iCS
iDBP
iDIS
iLBX
i$_m$
iMMX
Insite
int$_e$l
int$_e$lBOS
Intelevision
int$_e$ligent Identifier
int$_e$ligent Programming
Intellec
Intellink
iOSP
iPDS
iRMX
iSBC
iSBX
iSDM
iSXM
KEPROM
Library Manager
MCS
Megachassis
MICROMAINFRAME
MULTIBUS
MULTICHANNEL
MULTIMODULE
Plug-A-Bubble
PROMPT
Promware
QueX
QUEST
Ripplemode
RMX/80
RUPI
Seamless
SOLO
SYSTEM 2000
UPI

A1479 / 685 / 7K / DD / KH

SOFTWARE

REV.	REVISION HISTORY	DATE	APPD.
-001	Original issue.	2/81	
-002	Corrected errors in original manual.	10/81	
-003	Added information on WORD, LONGINT, LONGREAL, and TEMPREAL data types, new predefined functions, and segmentation controls.	3/82	
-004	Added information on virtual symbol table capacity, file I/O, iRMX systems, and miscellaneous corrections to text.	9/83	C.C.
-005	Updated to include information on data objects >64K, conditional conditional compilation, and iAPX 186 support.	2/85	D.L.N.

PREFACE

This manual gives instructions for programming in Pascal-86 and for using the Pascal-86 compiler to prepare programs for iAPX 86 and iAPX 88 microcomputer systems. It is primarily a reference manual for use when you are writing or compiling Pascal-86 programs; however, it also contains some introductory information to help you familiarize yourself with Pascal-86 as you start to use it.

The manual assumes you are familiar with basic programming concepts, including structured programming. However, it defines the language completely, assuming no prior knowledge of Pascal.

Following the description of the language, this manual provides instructions for compiling your Pascal programs, linking and locating the compiled code, and executing the final program. It explains how to interpret compiler output, including error messages. These portions assume you are familiar with the console operation of your development system.

Finally, the appendixes provide quick reference information, plus supplementary instructions for interfacing Pascal-86 modules to modules in other languages and to your own operating system software.

Manual Organization

Figure 0-1 illustrates the structure of this manual. As shown in the figure, it contains four kinds of information:

- Introductory and general reference information, including installation instructions
- Language information, for use when you are programming in Pascal-86
- Operating instructions for the compiler and run-time support, including descriptions of compiler controls
- Interfacing information you need if you supply some of your own systems software in place of that supplied by Intel (e.g., a non-Intel operating system or your own real arithmetic error handler), or if you are interfacing Pascal-86 modules to modules written in other languages such as ASM86 or PL/M-86

If you are a manager evaluating Pascal-86 to determine whether it fits your needs, you will find most of the information you need in Chapter 1, which is an overview of the product. You might also skim through Appendixes A through F for a summary of the language; note that the shaded portions of Appendixes D, E, and F describe extensions to ISO standard Pascal (Draft Proposal 7185).

To get started with Pascal-86, first read this preface (How to Use this Manual) and Chapter 1. (If you are familiar with assembly languages but not with high-level languages, see section 1.2.1 for a discussion of the advantages of a high-level language such as Pascal.) Then install the compiler using the instructions in Chapter 1, and try compiling, linking, locating, and running sample program 1 at the end of Chapter 1 to verify that the software operates correctly.

After that, if Pascal is a new language for you, study the sample programs in Chapter 9 and run some of them following the instructions in that chapter. Finally, skim through the manual from Chapter 2 to the end, and try writing and running a few programs of your own. Once you have become familiar with Pascal-86, you will

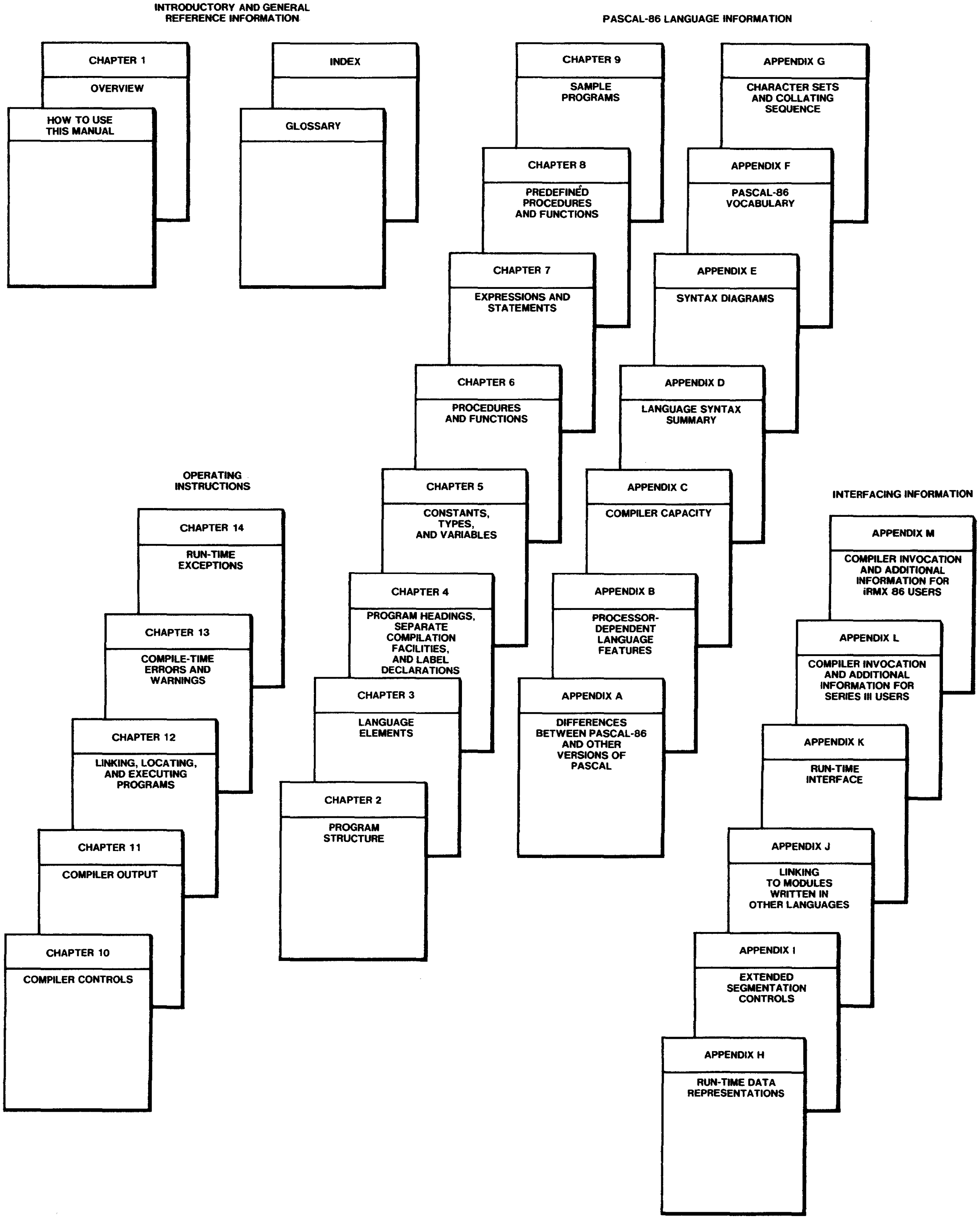

INTRODUCTORY AND GENERAL REFERENCE INFORMATION
CHAPTER 1
OVERVIEW
HOW TO USE THIS MANUAL
INDEX
GLOSSARY
PASCAL-86 LANGUAGE INFORMATION
CHAPTER 9
SAMPLE PROGRAMS
CHAPTER 8
PREDEFINED PROCEDURES AND FUNCTIONS
CHAPTER 7
EXPRESSIONS AND STATEMENTS
CHAPTER 6
PROCEDURES AND FUNCTIONS
CHAPTER 5
CONSTANTS, TYPES, AND VARIABLES
CHAPTER 4
PROGRAM HEADINGS, SEPARATE COMPILATION FACILITIES, AND LABEL DECLARATIONS
CHAPTER 3
LANGUAGE ELEMENTS
CHAPTER 2
PROGRAM STRUCTURE
APPENDIX G
CHARACTER SETS AND COLLATING SEQUENCE
APPENDIX F
PASCAL-86 VOCABULARY
APPENDIX E
SYNTAX DIAGRAMS
APPENDIX D
LANGUAGE SYNTAX SUMMARY
APPENDIX C
COMPILER CAPACITY
APPENDIX B
PROCESSOR-DEPENDENT LANGUAGE FEATURES
APPENDIX A
DIFFERENCES BETWEEN PASCAL-86 AND OTHER VERSIONS OF PASCAL
OPERATING INSTRUCTIONS
CHAPTER 14
RUN-TIME EXCEPTIONS
CHAPTER 13
COMPILE-TIME ERRORS AND WARNINGS
CHAPTER 12
LINKING, LOCATING, AND EXECUTING PROGRAMS
CHAPTER 11
COMPILER OUTPUT
CHAPTER 10
COMPILER CONTROLS
INTERFACING INFORMATION
APPENDIX M
COMPILER INVOCATION AND ADDITIONAL INFORMATION FOR iRMX 86 USERS
APPENDIX L
COMPILER INVOCATION AND ADDITIONAL INFORMATION FOR SERIES III USERS
APPENDIX K
RUN-TIME INTERFACE
APPENDIX J
LINKING TO MODULES WRITTEN IN OTHER LANGUAGES
APPENDIX I
EXTENDED SEGMENTATION CONTROLS
APPENDIX H
RUN-TIME DATA REPRESENTATIONS

Figure 0-1. Structure of This Manual 121539-24

find this manual useful as a complete reference. For quick reference, see the *Pascal-86 Pocket Reference.*

If you wish to transport existing Pascal programs to your iAPX 86 or iAPX 88 application system, refer to Appendix A for a list of the differences between Pascal-86 and other dialects of Pascal. This appendix indicates the areas of your programs that may require modification. If your programs are written completely in standard Pascal as defined by the ISO *Draft Proposal*, you need not modify them at all before you recompile them with Pascal-86.

Once you have coded your programs, you are ready to compile, link, locate, and run them. Refer to Chapter 10 for the use of compiler controls, and Chapter 11 for interpretation of the output listing. Chapter 12 describes how to link, locate, and execute your compiled programs. Chapters 13 and 14 help you interpret error messages you may receive when compiling or running your programs. Note that Chapter 12 gives only a brief outline of the linking and locating process and the associated error messages; for details, refer to the *iAPX 86, 88 Family Utilities User's Guide.*

If you are coding some of your application software in another language such as ASM86 or PL/M-86, refer to Appendixes H and J for the information you need. If you are interfacing to your own operating system or providing your own file/device drivers, refer to Appendix K for instructions.

How to Use This Manual

Section Numbers

All chapters and appendixes are section-numbered for easy cross-referencing: for instance, the heading number 5.3.2 denotes Chapter 5, section 3, subsection 2. When the text of one section refers to another section, the reference is made by number, for example, "as described in 7.1.3." Figures, tables, and sample programs are also numbered to aid in cross-referencing, for example, "in table 7-1," "see figure 9-5," "Sample Program 1 illustrates..."

Syntax Notation

This manual employs a notation similar to that used in Jensen and Wirth's *Pascal User Manual and Report* to define the syntax of the language precisely. The syntax of the entire Pascal-86 language, in this notation, is given in Appendix D. For those who prefer the syntax diagrams used in an appendix to the *Pascal User Manual* and in a number of textbooks on Pascal, Appendix E provides the syntax of the language in that form.

Notational Conventions

UPPERCASE Characters shown in uppercase must be entered in the order shown. You may enter the characters in uppercase or lowercase.

italic

Italic indicates a meta symbol that may be replaced with an item that fulfills the rules for that symbol. The actual symbol may be any of the following:

directory-name

Is that portion of a *pathname* that acts as a file locator by identifying the device and/or directory containing the *filename.*

filename

Is a valid name for the part of a *pathname* that names a file.

pathname

Is a valid designation for a file; in its entirety, it consists of a *directory* and a *filename.*

pathname1,
pathname2, ...

Are generic labels placed on sample listings where one or more user-specified pathnames would actually be printed.

system-id

Is a generic label placed on sample listings where an operating system-dependent name would actually be printed.

V*x.y*

Is a generic label placed on sample listings where the version number of the product that produced the listing would actually be printed.

[ ]

Brackets indicate optional arguments or parameters.

{ }

One and only one of the enclosed entries must be selected unless the field is also surrounded by brackets, in which case it is optional.

{ }...

At least one of the enclosed items must be selected unless the field is also surrounded by brackets, in which case it is optional. The items may be used in any order unless otherwise noted.

|

The vertical bar separates options within brackets [ ] or braces { }.

...

Ellipses indicate that the preceding argument or parameter may be repeated.

[,...]

The preceding item may be repeated, but each repetition must be separated by a comma.

punctuation

Punctuation other than ellipses, braces, and brackets must be entered as shown. For example, the punctuation shown in the following command must be entered:

```
SUBMIT PLM86(PROGA,SRC,'9 SEPT 81')
```

input lines

In interactive examples, user input lines are printed in white on black to differentiate them from system output.

<cr>

Indicates a carriage return.

CAUTION

Caution.

shading — Intel extensions to standard Pascal and descriptions of the extensions are shaded in gray.

When two adjacent items must be concatenated, they appear with no space between them. A blank space between two items indicates that the two items may be separated by one or more *logical-blanks*. For example:

digits.digits [E [*sign*]*digits*] <cr>

specifies that the first set of *digits*, the **.** symbol, and the second set of *digits* must be concatenated, with no blanks between them. Likewise, the **E** symbol, the *sign* if included, and the third set of *digits* must be concatenated.

Alternative constructs are represented as vertically adjacent items separated by extra vertical spacing and enclosed between curly braces that are taller than a single line of type. When these braces appear, choose any one of the constructs enclosed between the braces. For example:

{ *digits*
binary-digit [*binary-digit*] . . . B
octal-digit [*octal-digit*] . . . Q
digit [*hex-digit*] . . . H }

indicates that the construct described may have any one of the four forms listed between the large braces.

Text enclosed between the character sequence (* and the sequence *), when these symbols are in light, non-monospace type, is a prose definition of the given construct. Such definitions are used when symbolic definitions would be more cumbersome. For example:

(* any uppercase or lowercase letter of the alphabet *)

is used to avoid listing 52 separate characters vertically between braces.

The start of a new line in the notation does not mean you must start a new line at that point in your program; however, you may do so for readability. For example, when you use the construct:

```
FOR variable :=expression TO expression
    DO statement
```

you need not include a carriage return after the second *expression*, but in many programs doing so makes the statement more readable.

TABLE OF CONTENTS

CONTENTS

CHAPTER 1 OVERVIEW

PAGE

CHAPTER 2 PROGRAM STRUCTURE

CHAPTER 3 LANGUAGE ELEMENTS

CHAPTER 4 PROGRAM HEADINGS, SEPARATE COMPILATION FACILITIES, AND LABEL DECLARATIONS

CHAPTER 5 CONSTANTS, TYPES, AND VARIABLES

PAGE

CHAPTER 6 PROCEDURES AND FUNCTIONS

CHAPTER 7 EXPRESSIONS AND STATEMENTS

CHAPTER 8 PREDEFINED PROCEDURES AND FUNCTIONS

PAGE

CHAPTER 9
SAMPLE PROGRAMS

CHAPTER 10
COMPILER CONTROLS

CHAPTER 11
COMPILER OUTPUT

PAGE

CHAPTER 12
LINKING, LOCATING, AND EXECUTING PROGRAMS

CHAPTER 13
COMPILE-TIME ERRORS AND WARNINGS

CHAPTER 14
RUN-TIME EXCEPTIONS

APPENDIX A
DIFFERENCES BETWEEN PASCAL-86 AND OTHER VERSIONS OF PASCAL

APPENDIX B
PROCESSOR-DEPENDENT LANGUAGE FEATURES

APPENDIX C
COMPILER CAPACITY

APPENDIX D
LANGUAGE SYNTAX SUMMARY

PAGE

APPENDIX E
SYNTAX DIAGRAMS

APPENDIX F
PASCAL-86 VOCABULARY

APPENDIX G
CHARACTER SETS AND COLLATING SEQUENCE

APPENDIX H
RUN-TIME DATA REPRESENTATIONS

APPENDIX I
PASCAL-86 EXTENDED SEGMENTATION MODELS

APPENDIX J
LINKING TO MODULES WRITTEN IN OTHER LANGUAGES

APPENDIX K
RUN-TIME INTERFACE

PAGE

APPENDIX L
COMPILER INVOCATION AND ADDITIONAL INFORMATION FOR SERIES III USERS

APPENDIX M
COMPILER INVOCATION AND ADDITIONAL INFORMATION FOR iRMX™ 86 USERS

GLOSSARY

INDEX

FIGURES

FIGURE TITLE PAGE

TABLES

CHAPTER 1
OVERVIEW

This chapter introduces Pascal-86 and explains how it fits into the process of developing software for your iAPX 86 or iAPX 88 application system.

1.1 Product Definition

Pascal-86 is a high-level language designed for programming the iAPX 86, 88 family of microprocessors. It is a superset of standard Pascal as defined in the ANSI/IEEE770X3.97–1983 and includes additional features useful in microprocessor applications.

The Pascal-86 compiler translates your Pascal-86 source programs into relocatable object modules, which you can then link to other such modules, coded in Pascal or in other iAPX 86, 88 languages. The compiler provides listing output, error messages, and a number of compiler controls to aid in program development and debugging.

With the compiler comes a set of relocatable object libraries to be linked in with your own code; these libraries provide complete run-time support, including input/output and an optional interface to the Intel 8087 Numeric Data Processor to optimize arithmetic operations. After linking your own modules together with these Intel-supplied library modules, you can locate your final linked program to run on an Intellec development system, or in RAM, PROM, or ROM in your own iAPX 86 or iAPX 88 microcomputer system.

To perform the steps following compilation, use the 8086-based iAPX 86, 88 Family software development utilities — LINK86, LIB86, LOC86, CREF86, and OH86. You debug your programs using the DEBUG-86 applications debugger, PSCOPE (the interactive symbolic debugger), or the ICE-86A or ICE-88 In-Circuit Emulator. For firmware systems, you then use the Universal PROM Programmer (UPP) with its Universal PROM Mapper (UPM) software to transfer your programs to PROM.

1.2 The Pascal-86 Language

1.2.1 Using a High-Level Language

High-level languages (Pascal in particular) more closely model the human thought process than do lower-level languages, such as assembly language. They therefore are easier and faster to write, since one less translation step is required from concept to code. High-level language programs are also more likely to be correct, since there is less occasion to introduce error.

Programs in a high-level language are easier to read and understand, and thus easier to modify. As a result, you can develop high-level language programs in a much shorter period of time, and these programs are easier to maintain throughout the life of the product. Thus high-level languages result in lower costs for both development and maintenance of programs.

In addition, programs in a high-level language, particularly a standardized language like Pascal, are easily transferred from one processor to another. Programs that can be transferred between processors without modification are said to be *portable.*

As you might expect, these advantages have a price: the resulting translated machine programs normally require more memory space and may run more slowly. For this reason, after the initial software design is complete, you may wish to re-code your most time-critical and space-critical routines in assembly language.

If Pascal-86 is your first high-level language, you probably want to know how programming in a high-level language differs from assembly-language programming. When you use a high-level language:

- You do not need to know the instruction set of the processor you are using.
- You need not be concerned with the details of the target processor, such as register allocation or assigning the proper number of bytes for each data item—the compiler takes care of these things automatically.
- You use keywords and phrases that are closer to natural English.
- You can combine many operations (including arithmetic, Boolean, and set operations) into expressions; thus you can perform a whole sequence of operations with one statement.
- You can use data types and data structures that are closer to your actual problem; for instance, in Pascal you can program in terms of Boolean variables, characters, arrays, and files rather than bytes and words.

The introductory example at the end of this chapter (section 1.7) illustrates these points. Compare this Pascal program with an assembly-language program you might write to solve the same problem.

Coding programs in a high-level language involves thinking at a different level than coding in assembly language. This level is closer to the level of thinking you use when you are planning your overall system design.

1.2.2 Why Pascal?

Many high-level programming languages are available today; some of them have been around far longer than Pascal. So once you have decided to use a high-level language, your next questions may be: How does Pascal differ from other high-level languages? What advantages does it have? When is it the right language to use?

Here are some of the characteristics of Pascal:

- It has a block structure similar to that of PL/M, plus control constructs that aid—in fact, encourage and enforce—structured programming.
- It includes facilities for such data structures as multi-dimensional arrays, records, sets, files, and pointer-based dynamic variables, and also allows you to define new data types related to your problem, e.g., weekday, patientrecord.
- It is a strongly typed language—that is, the compiler does extensive data type compatibility checking and range checking to help you detect logic errors in your programs at compile time.
- It includes run-time support for sequential file I/O and floating-point arithmetic.
- Its data structuring facilities and control statements are designed in a logically consistent way. Thus Pascal is a particularly good language for expressing algorithms, and has been used for this purpose in many textbooks.
- Its control constructs make program correctness relatively easy to verify.
- It is a standard language used on many computers, so Pascal programs are portable.

For iAPX 86 and 88 systems, Intel offers Pascal, PL/M, and FORTRAN. Your choice among these should depend on your implementation. Pascal-86, with its run-time I/O support, data-structuring facilities, user-defined types, and special-purpose built-in procedures and functions, is a higher-level language than PL/M-86, and is therefore better suited to applications programming. Pascal-86 also provides more extensive type checking than either PL/M-86 or FORTRAN-86, thus reducing program debugging time. Because Pascal is a standard language, programs in Pascal are portable — they can be used on a number of different processors. On the other hand, PL/M-86 allows you to program at a level closer to your microprocessor hardware, making it generally more suitable for systems programming, while FORTRAN-86 has a rich set of arithmetic operations which make it best suited to scientific and numerical applications.

The philosophy behind the Pascal and PL/M languages is fundamentally different. Pascal's strong typing and other language features impose a strict discipline on you, the programmer, to enforce good structured programming practice and help you detect errors in your programs. Certain programming practices that make errors hard to find — such as defining one data type on top of another — are forbidden in Pascal. PL/M, on the other hand, was designed for programmers (generally systems programmers) who need such features and are willing to take the risk and extra debugging time required by programs that use them.

What about the differences between Pascal and older, better-established languages like BASIC and COBOL Pascal has many more features than BASIC; and thanks to more consistent standardization, it is also more portable. It is a more general-purpose language than COBOL, which is tailored for business data processing. In addition, Pascal differs from these other languages in its strong typing and block structure.

Pascal was designed in 1973 by Niklaus Wirth, who had two main objectives: to produce a language suitable for teaching programming concepts, and to design that language so that its implementations on existing computers could be reliable and efficient. Wirth found the more traditional languages (including FORTRAN, COBOL, and PL/I) unsuitable for teaching: their features and constructs often cannot be explained logically, making them more difficult to learn.

Even more important, he was convinced that the language in which a programmer learns to program profoundly influences his thinking, and therefore his programming style and his reasoning in problem solving. He concluded that teaching programming using a logically constructed language can lead to better programmers and better programs.

Pascal's principles of structuring and form of expressions were patterned after those of Algol 60. However, other constructs were changed from Algol to accommodate Pascal's additional data structuring facilities. Record and file structures more useful for solving commercial-type problems were added to Pascal.

1.2.3 Portability

As mentioned earlier, Pascal-86 conforms to standard Pascal as defined in the ISO *Draft Proposal*. This means that you can take Pascal programs written for other processors, compile them using Pascal-86, and run them on an iAPX 86 or iAPX 88 microcomputer system, provided you use only standard features. The same programs can run on iAPX 86 systems and iAPX 88 systems without change.

You can write complete programs in Pascal-86 without using any Intel extensions to standard Pascal, thus keeping your programs completely portable. In this manual, the descriptions of Intel extensions are shaded in gray to distinguish them from standard features. You can also use a compiler option (the NOEXTENSIONS control) to direct the compiler to print out warning messages wherever such extensions appear in your program.

1.2.4 Intel Extensions to Standard Pascal

If you are concerned with the ease of programming for your microprocessor applications, you will probably want to use the language extensions and compiler controls that tailor Pascal-86 to the iAPX 86, 88 environment. These include 32-bit arithmetic, language constructs for building separately compiled modules, and builtin procedures for port input/output and interrupt control.

Separately compiled modules allow you to divide a program into smaller, more manageable parts, and to locate different parts of your program in memory of different types or in different hardware locations. Port input/output provides fast data transfer by means of direct communication with microprocessor ports. The interrupt control procedures allow you to write Pascal routines to handle interrupts in your system.

1.3 The Compiler and Run-Time System

1.3.1 Compiler Features

The Pascal-86 compiler includes a number of features to make programming and debugging easier. Compiler controls allow you to specify the form and content of your source code, object code, and output listing.

Controls are provided to copy (INCLUDE) source code from other files in addition to the main source file, to output type and debug information in the object file for use by LINK86 and the ICE-86A and ICE-88 emulators, and to specify interrupt procedures tailored to your hardware. The compiler also provides the NOEXTENSIONS control to flag extensions to standard Pascal, an optional cross-reference listing, and a control to aid in program checkout and debugging.

1.3.2 Run-Time Support Libraries

The run-time support libraries, provided in relocatable object code form to be linked to your compiled object program, allow you to run your program in a number of environments. You simply choose the run-time libraries that match the hardware/software configuration you are using.

These libraries provide all I/O support needed to run your programs. You may also choose to have floating-point arithmetic operations performed using either floating-point software routines on your 8086 or 8088 processor, or the on-chip capabilities of an 8087 Numeric Data Processor for higher performance; in either case, all required arithmetic and interface software is included in the run-time libraries. In addition, the modular structure of these libraries allows you to substitute your own file/device drivers.

1.4 Hardware and Software Environments

1.4.1 Program Development Environment

To run the compiler, you must have the following hardware and software:

- Intellec Series III development system and resident operating system (see Appendix L).
- 86/300 Microcomputer System (see Appendix M).
- Custom iAPX 86 or iAPX 88 Microcomputer System that includes an iRMX 86-based resident operating system (see Appendix M).
- Four double-density diskette drives or a hard disk unit is recommended. (Note that for hard disk users, initial installation of the compiler requires a single- or double-density diskette drive, since the product is delivered in diskette form; thereafter, hard disk alone is sufficient.)

A system with a diskette or hard disk unit and a printer is also recommended for producing hard-copy output listings. This system may be separate from the one used to compile programs.

To link and relocate programs after you have compiled them, and to prepare them for loading (or PROM programming) and execution, you need the following 8086-based software:

- LINK86
- LIB86
- LOC86
- CREF86
- OH86

Instructions for using these utility programs are given in the *iAPX 86, 88 Family Utilities User's Guide*, Order Number 121616.

Depending on your development environment and your final run-time environment, you may also wish to use the following hardware and software:

- The DEBUG-86 applications debugger
- The PSCOPE symbolic debugger
- The ICE-86A or ICE-88 In-Circuit Emulator
- The SDK-86 System Design Kit, optionally with the SDK-C86 Software and Cable Interface
- The iSBC 957A Intellec-iSBC 86/12A Interface and Execution Package
- The Universal PROM Programmer (UPP) with the Universal PROM Mapper (UPM) software

1.4.2 Run-Time Environment

Your compiled, linked, and located program code may run in either of the following environments:

- A Series III development system with its ISIS-II based resident operating system
- An iSBC 86-based system with an iSBC 86-based single board computer or a custom-designed iAPX86 or iAPX88 microcomputer system.

In the latter case (an environment without Intel operating system support), you will need to provide your own operating system support for the run-time libraries. Appendix K gives instructions for writing your own file/device drivers and the software interface required by the run-time libraries.

In the iRMX 86-based software run-time environment, the Universal Development System Interface layer must be configured into iRMX 86 in order to run PASCAL-86 programs.

You may increase the speed of floating-point arithmetic operations in your programs by including an 8087 Numeric Data Processor in your system. Detailed specifications are provided in the *iAPX 86,88 User's Manual*, Order Number 210201-001.

1.5 Compiler Installation

The Pascal-86 software package includes this manual (the *Pascal-86 User's Guide*), the *Pascal-86 Pocket Reference*, Order Number 121541, supplementary literature including a customer letter and Problem Report forms, and one double- and two single-density program diskettes. The diskettes contain the following files:

PASC86.86
P86RN0.LIB
P86RN1.LIB
P86RN2.LIB
P86RN3.LIB
RTNULL.LIB
DCON87.LIB
CEL87.LIB
EH87.LIB
E8087
E8087.LIB
8087.LIB
87NULL.LIB
PROG1.SRC
PROG2A.SRC
PRG2B1.SRC
PRG2B2.SRC
PROG3.SRC
PROG4.SRC
PROG5.SRC
PROG6.SRC
PROG7.SRC
PROG8.SRC
PROG9.SRC
DATA2
DATA3
DATA4

The file named PASC86.86 contains the Pascal-86 compiler. The files P86RN0.LIB, P86RN1.LIB, P86RN2.LIB, P86RN3.LIB, RTNULL.LIB, DCON87.LIB, CEL87.LIB, EH87.LIB, E8087, E8087.LIB, 8087.LIB, and 87NULL.LIB are the run-time support libraries and modules. (Detailed descriptions of the 8087 libraries are located in the *8087 Support Library Reference Manual* Order Number 121725.) PROG1.SRC, PROG2A.SRC, PRG2B1.SRC, PRG2B2.SRC, PROG3.SRC, PROG4.SRC, PROG5.SRC, PROG6.SRC, PROG7.SRC, PROG8.SRC, and PROG9.SRC are the source code for the sample programs in Chapters 1, 2, and 9. DATA2, DATA3, and DATA4 are the data files for the sample programs.

NOTE

In the iRMX 86 environment the ".86" extension is dropped.

Once you have your compile-time environment configured as described in section 1.4.1, copy the compiler and run-time library files from the release diskette to the single- or double-density diskette or hard disk you are using on your system.

The sample programs provided on the release diskette may be used for demonstration and checkout in your development environment. Operating instructions for these programs are given in Chapter 9.

1.6 The Program Development Process

The Pascal-86 compiler and run-time libraries are part of an integrated set of tools that make up the total iAPX 86 or iAPX 88 development solution for your microcomputer system. Figure 1-1 shows how you use these tools to develop programs. The shaded boxes represent Intel products.

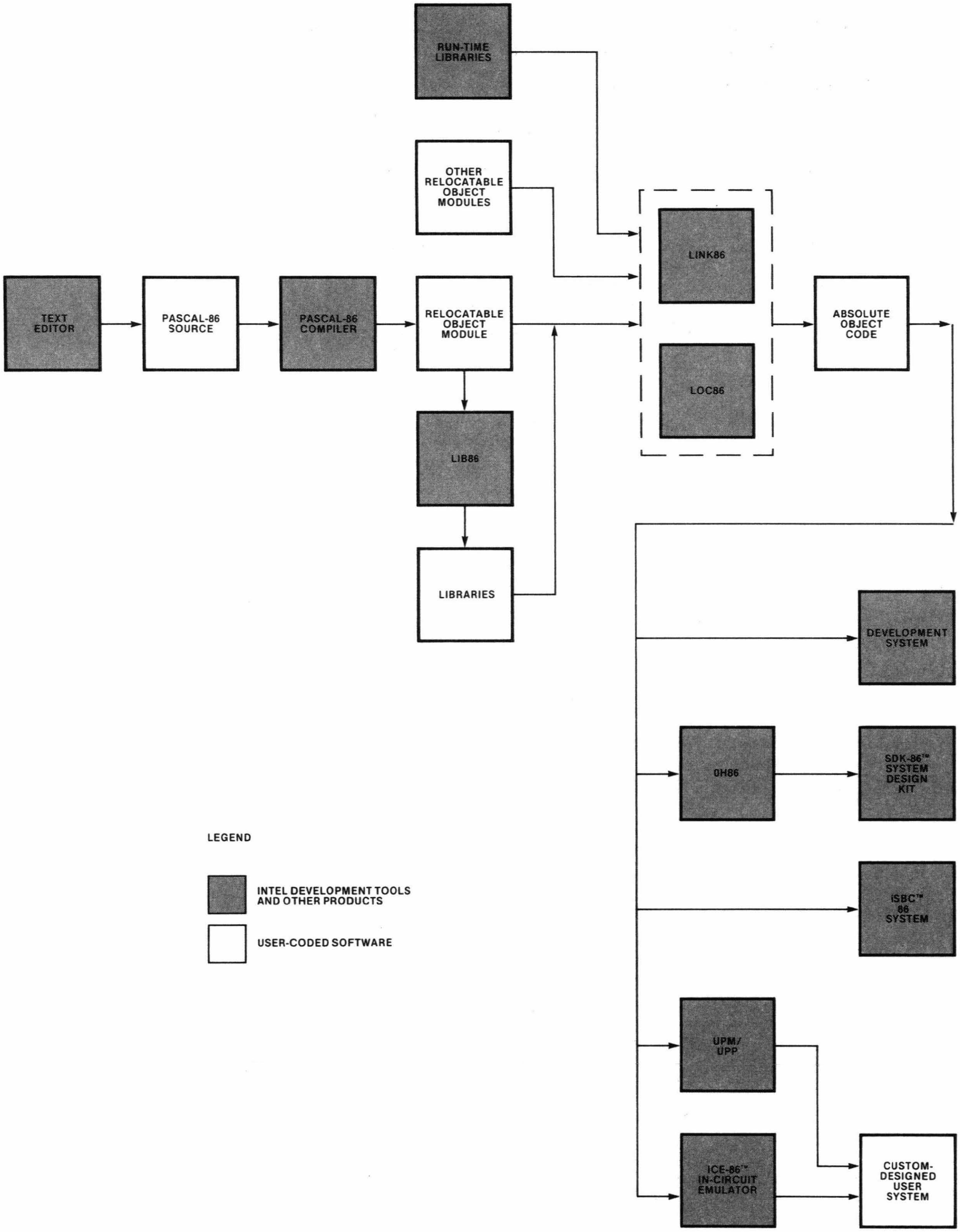

RUN-TIME LIBRARIES
OTHER RELOCATABLE OBJECT MODULES
TEXT EDITOR
PASCAL-86 SOURCE
PASCAL-86 COMPILER
RELOCATABLE OBJECT MODULE
LIB86
LIBRARIES
LINK86
LOC86
ABSOLUTE OBJECT CODE
DEVELOPMENT SYSTEM
OH86
SDK-86™ SYSTEM DESIGN KIT
iSBC™ 86 SYSTEM
UPM/ UPP
ICE-86™ IN-CIRCUIT EMULATOR
CUSTOM-DESIGNED USER SYSTEM
LEGEND
INTEL DEVELOPMENT TOOLS AND OTHER PRODUCTS
USER-CODED SOFTWARE

Figure 1-1. Pascal-86 Program Development Process

121539-69

The steps in the software development process are as follows:

1. Define the problem completely.
2. Outline the proposed solution in terms of hardware plus software. Once this step is done, you may begin designing your hardware.
3. Design the software for your system. This important step may consist of several sub-steps, including breaking down the task into modules, choosing the programming language, and selecting the algorithms to be used.
4. Code your programs and prepare them for translation using a text editor.
5. Translate your Pascal program code using the Pascal-86 compiler.
6. Using the text editor, correct any compile-time errors; then recompile.
7. Using 8086-based LINK86 (and LOC86 if needed), link the resulting relocatable object module to the necessary run-time libraries supplied with Pascal-86, and locate your object code. The use of LINK86 and LOC86 depends on your application; for detailed instructions, see the *iAPX 86, 88 Family Utilities User's Guide.*
8. You can then run your programs and debug them, with the aid of run-time error messages and diagnostic output generated by the compiler's program checkout control. Your execution vehicle for debugging can be an operating system with the DEBUG-86 or PSCOPE applications debugger and an ICE-86A or ICE-88 In-Circuit Emulator, or RAM on an SDK-86 System Design Kit or iSBC 86/12A Single Board Computer with resident monitor.
9. Translate and debug your other system modules, including those coded in other languages. Once you have performed the desired amount of testing on your individual modules, you can link them together and locate them using 8086-based LINK86 and LOC86.
10. Test and debug your software in your chosen debugging environment (see step 8).
11. Produce a final debugged object module and transfer it to your run-time environment. How you do this depends on the nature of that environment and the tools you are using.

    - If it is a Series III, use the Series III RUN command to load and run your program.
    - If it is RAM on an SDK-86 kit or an iSBC 86 Single Board Computer system, use OH86 to obtain a hexadecimal object code file. Then, if you have been developing your programs on a Series III, use an appropriate tool for downloading them into your execution board (the ICE-86A or ICE-88 In-Circuit Emulator, the SDK-C86 Software and Cable Interface, or the iSBC 957 Interface and Execution Package).
    - If it is ROM on an SDK-86 kit, iSBC Single Board Computer system, or your own custom-designed hardware, use the Universal PROM Programmer (UPP) with its Universal PROM Mapper (UPM) software to transfer your program to PROM.

Note that you can do your hardware development in parallel with software development, and that you can take intermediate hardware/software integration steps if you are using the ICE-86A or ICE-88 In-Circuit Emulator.

For instructions on the use of other Intel products discussed in this section, refer to the manuals listed in your specific operating-system appendix.

1.7 An Introductory Sample Program

Figure 1-2 is a Pascal-86 program that converts Fahrenheit temperatures to Celsius as you enter them from the console. The source code for this program is provided on the release diskette as the file named PROG1.SRC. This section explains, step by step, how to compile, link, and run the program on your development system.

NOTE

> This introductory sample program is intentionally an extremely simple one. Larger sample programs appear in Chapters 2 and 9.

The interactive computer dialogue in this section consists of commands you enter, which are immediately echoed on the console display, and text displayed by the operating system, the Pascal-86 compiler, and other Intel-supplied programs. The text you enter is shown in reverse type (white on a black background), and the text displayed by the Intel programs is shown in normal black type. The notation **<** cr **>** stands for the RETURN key on the console keyboard. Note that the operating system prompt (indicating that it is ready to accept a command) and, for some systems, the name of the loader (e.g., RUN on the Series III) are *not* included—see your specific operating-system appendix for details. The two-asterisk prompt indicating the beginning of a continuation line is given here.

To prepare this sample program for execution on your operating system, make a copy of the file PROG1.SRC. (If this were your own program, you would first type it in using a text editor.) You can invoke the compiler using the command:

```
-run pasc86 :f5:prog1.src
```

The compiler responds on the console with a sign-on message:

```
system-id Pascal-86, Vx.y
Copyright 1981, 1982, 1983 Intel Corporation
```

where

system-id is the name of your operating system.

x.y is the version number of the compiler.

As the compiler processes the program, a trace of the various compilation phases is displayed below the sign-on message. For this example, the final completed trace line is:

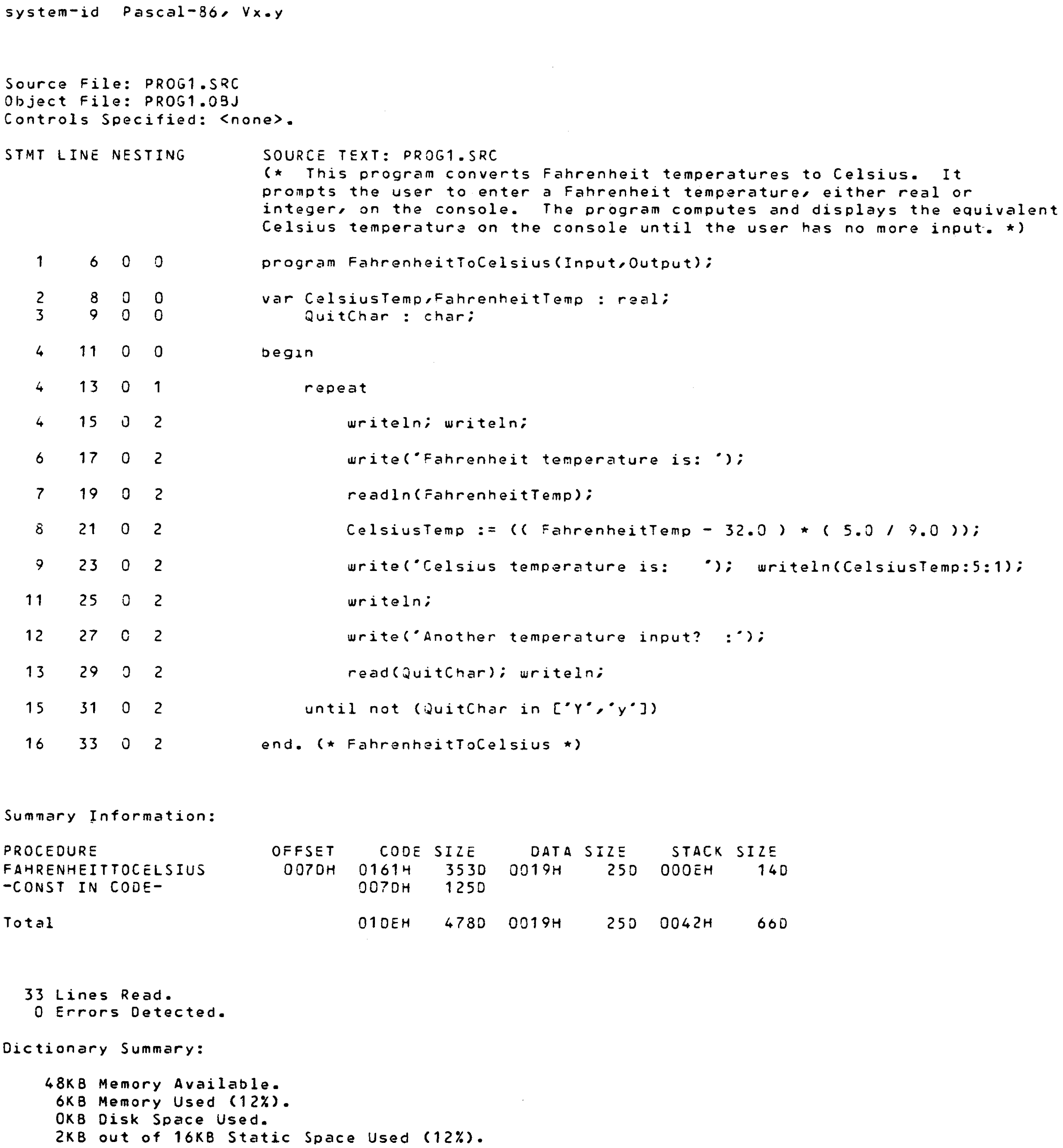
```
PARSE(0), ANALYZE(0), NOXREF, OBJECT
```

This is followed by the console sign-off message:

```
Compilation of FAHRENHEITTOCELSIUS Completed, 0 Errors Detected.
End of Pascal-86 Compilation.
```

Next, link the resulting object program with the necessary run-time libraries. To do this, enter the following command:

```
LINK86 PROG1.OBJ, P86RN0.LIB, P86RN1.LIB, &<cr>
**P86RN2.LIB, P86RN3.LIB, E8087.LIB, E8087, &<cr>
**LARGE.LIB TO PROG1.86 BIND<cr>
```



```
system-id  Pascal-86, Vx.y


Source File: PROG1.SRC
Object File: PROG1.OBJ
Controls Specified: <none>.

STMT LINE NESTING       SOURCE TEXT: PROG1.SRC
                        (*  This program converts Fahrenheit temperatures to Celsius.  It
                        prompts the user to enter a Fahrenheit temperature, either real or
                        integer, on the console.  The program computes and displays the equivalent
                        Celsius temperature on the console until the user has no more input. *)

   1    6  0  0         program FahrenheitToCelsius(Input,Output);

   2    8  0  0         var CelsiusTemp,FahrenheitTemp : real;
   3    9  0  0             QuitChar : char;

   4   11  0  0         begin

   4   13  0  1             repeat

   4   15  0  2                 writeln; writeln;

   6   17  0  2                 write('Fahrenheit temperature is: ');

   7   19  0  2                 readln(FahrenheitTemp);

   8   21  0  2                 CelsiusTemp := (( FahrenheitTemp - 32.0 ) * ( 5.0 / 9.0 ));

   9   23  0  2                 write('Celsius temperature is:   '); writeln(CelsiusTemp:5:1);

  11   25  0  2                 writeln;

  12   27  0  2                 write('Another temperature input?  :');

  13   29  0  2                 read(QuitChar); writeln;

  15   31  0  2             until not (QuitChar in ['Y','y'])

  16   33  0  2         end. (* FahrenheitToCelsius *)


Summary Information:

PROCEDURE                OFFSET    CODE SIZE      DATA SIZE     STACK SIZE
FAHRENHEITTOCELSIUS       0070H   0161H    353D  0019H     25D  000EH     14D
-CONST IN CODE-                   0070H    125D

Total                             010EH    478D  0019H     25D  0042H     66D


  33 Lines Read.
   0 Errors Detected.

Dictionary Summary:

    48KB Memory Available.
     6KB Memory Used (12%).
     0KB Disk Space Used.
     2KB out of 16KB Static Space Used (12%).
```

Figure 1-2. Sample Program 1: Temperature Conversion

LINK86 displays the sign-on message:

```
system-id 8086 LINKER, Vx.y
```

then links your program, returning control to the operating system when it finishes.

To run the program, first give the command:

```
PROG1<cr>
```

The program displays the message:

```
Fahrenheit temperature is:
```

Type in a temperature in Fahrenheit degrees. If you mis-type, you may edit the line using the RUBOUT key. Then strike the RETURN key.

The program calculates the Celsius temperature and displays the output:

```
Celsius temperature is: n
```

where

n is the Celsius equivalent of the temperature you typed in.

Finally, the program skips a line and displays:

```
Another temperature input? :
```

Type *Y* or *y* if you want to do another calculation. This causes the program to skip a space and display the starting message again, allowing you to type in another temperature. You may do this as many times as you wish.

When you wish to stop, answer the final query with any character other than *Y* or *y*, and the program will skip a line and return control to the operating system.

CHAPTER 2
PROGRAM STRUCTURE

Before we begin defining the specific rules for coding declarations, statements, and other language constructs in Pascal-86, let us look at the overall structure of a Pascal-86 program. This chapter provides a frame of reference for the six chapters that follow it (Chapters 3 through 8), which fill in the details of the language.

The features of Pascal are extremely interdependent. For this reason, this manual will sometimes need to refer to terms or concepts, such as statements, variables, and types, before it has defined or discussed them. If Pascal is a new language for you and you are reading this manual for the first time, you do not need to understand every concept thoroughly the first time you encounter it. Concepts that are mentioned briefly will be explained more fully in subsequent chapters.

Even if you are familiar with the block structure of standard Pascal, you should read this chapter before you start programming. It introduces the concept of a separately compiled module—an Intel extension to standard Pascal—and shows how the module construct fits in with standard Pascal program structure.

2.1 Structure of a Standard Pascal Program

Pascal is a *block-structured language*. Programs in such a language are composed of sections (called blocks) that perform logically related functions and may be nested inside one another. PL/M, PL/I, and Algol are also block-structured languages; FORTRAN, BASIC, COBOL, and most assembly languages are not. Block structure in a programming language has several advantages:

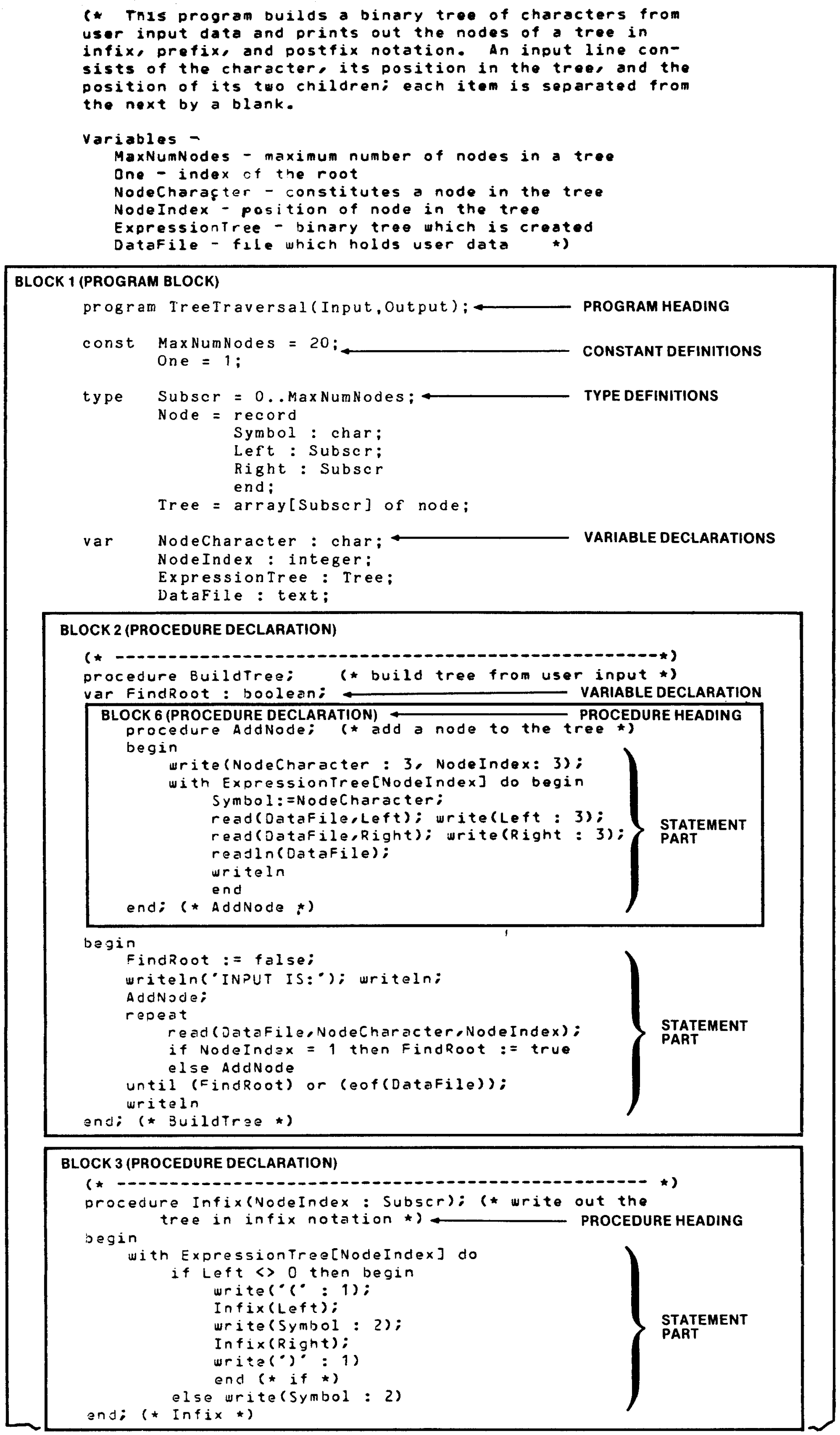
- It permits you to concentrate on one part of a program at a time, isolating that part from the rest of the program.
- It results in programs whose logical structure is easy to read and understand.
- It allows the compiler to impose certain rules and checks over the scope of variables and the flow of control in a program, enabling you to discover and correct many types of logical errors at compile time.

Figure 2-1 illustrates the structure of a Pascal program. (Section 9.2 gives a more detailed explanation of the sample program in figure 2-1.)

A *block* in Pascal consists of the following three parts:

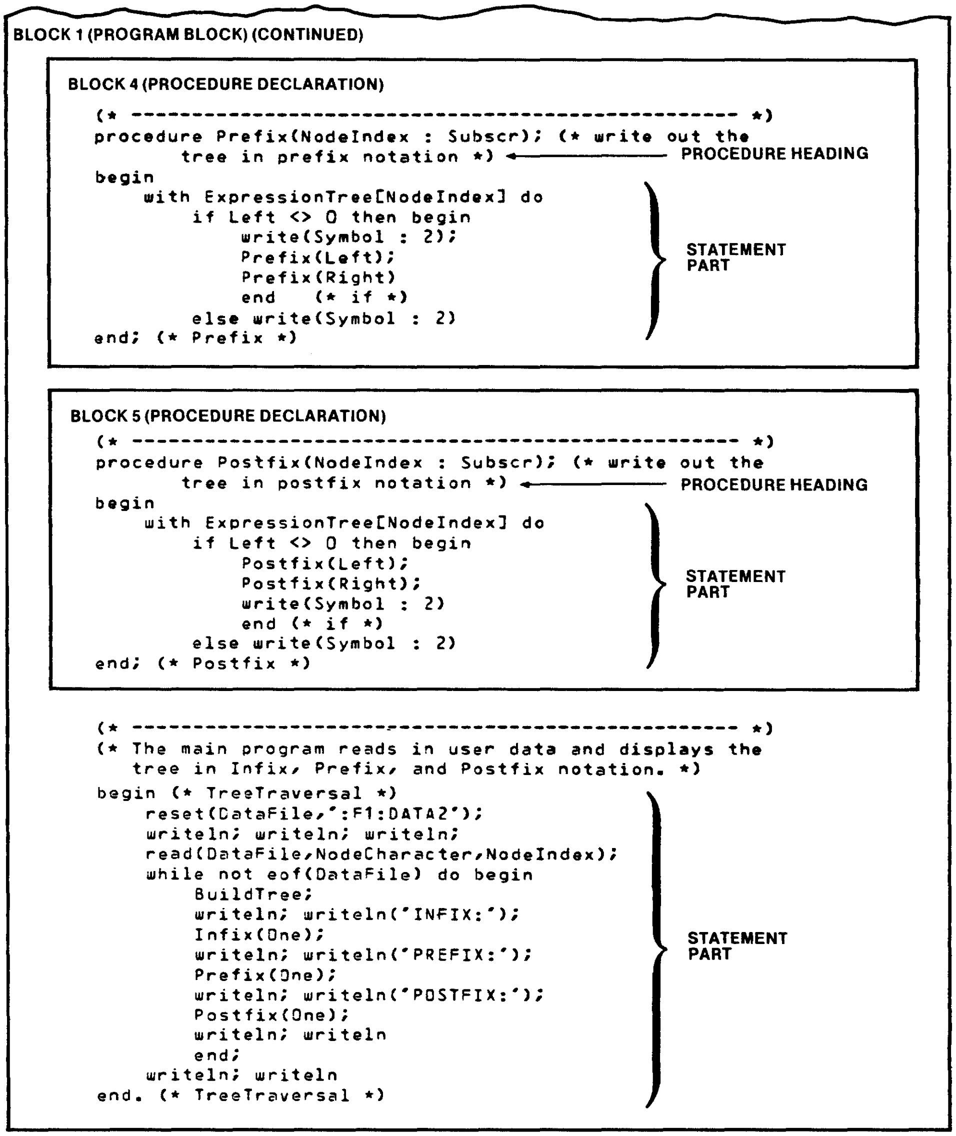
1. Definitions and declarations
2. Procedure and function declarations
3. Statements

The first part defines data items—constants, types, and variables—and labels used within the block. *Definitions* introduce items that have meaning only at compile time, and *declarations* introduce items that have meaning at run time as well. *Procedure declarations* and *function declarations* define subprograms, which are blocks *nested*, or contained, within the block in question. *Statements* specify the actions to be performed by the block. Of the three parts, only the statement part is required in every block.



```
(*  This program builds a binary tree of characters from
user input data and prints out the nodes of a tree in
infix, prefix, and postfix notation.  An input line con-
sists of the character, its position in the tree, and the
position of its two children; each item is separated from
the next by a blank.

Variables -
   MaxNumNodes - maximum number of nodes in a tree
   One - index of the root
   NodeCharacter - constitutes a node in the tree
   NodeIndex - position of node in the tree
   ExpressionTree - binary tree which is created
   DataFile - file which holds user data    *)

program TreeTraversal(Input,Output);

const  MaxNumNodes = 20;
       One = 1;

type   Subscr = 0..MaxNumNodes;
       Node = record
              Symbol : char;
              Left : Subscr;
              Right : Subscr
              end;
       Tree = array[Subscr] of node;

var    NodeCharacter : char;
       NodeIndex : integer;
       ExpressionTree : Tree;
       DataFile : text;

(* ---------------------------------------------------*)
procedure BuildTree;     (* build tree from user input *)
var FindRoot : boolean;
    procedure AddNode;  (* add a node to the tree *)
    begin
        write(NodeCharacter : 3, NodeIndex: 3);
        with ExpressionTree[NodeIndex] do begin
            Symbol:=NodeCharacter;
            read(DataFile,Left); write(Left : 3);
            read(DataFile,Right); write(Right : 3);
            readln(DataFile);
            writeln
            end
    end; (* AddNode *)
begin
    FindRoot := false;
    writeln('INPUT IS:'); writeln;
    AddNode;
    repeat
        read(DataFile,NodeCharacter,NodeIndex);
        if NodeIndex = 1 then FindRoot := true
        else AddNode
    until (FindRoot) or (eof(DataFile));
    writeln
end; (* BuildTree *)

(* ----------------------------------------------- *)
procedure Infix(NodeIndex : Subscr); (* write out the
       tree in infix notation *)
begin
    with ExpressionTree[NodeIndex] do
        if Left <> 0 then begin
            write('(' : 1);
            Infix(Left);
            write(Symbol : 2);
            Infix(Right);
            write(')' : 1)
            end (* if *)
        else write(Symbol : 2)
end; (* Infix *)
```

Figure 2-1. Sample Program 2A: Binary Tree Traversal in Standard Pascal



BLOCK 1 (PROGRAM BLOCK) (CONTINUED)

BLOCK 4 (PROCEDURE DECLARATION)

```
(* ------------------------------------------------------ *)
procedure Prefix(NodeIndex : Subscr); (* write out the
        tree in prefix notation *)
begin
    with ExpressionTree[NodeIndex] do
        if Left <> 0 then begin
            write(Symbol : 2);
            Prefix(Left);
            Prefix(Right)
            end   (* if *)
        else write(Symbol : 2)
end; (* Prefix *)
```

BLOCK 5 (PROCEDURE DECLARATION)

```
(* ------------------------------------------------------ *)
procedure Postfix(NodeIndex : Subscr); (* write out the
        tree in postfix notation *)
begin
    with ExpressionTree[NodeIndex] do
        if Left <> 0 then begin
            Postfix(Left);
            Postfix(Right);
            write(Symbol : 2)
            end (* if *)
        else write(Symbol : 2)
end; (* Postfix *)
```

```
(* ------------------------------------------------------ *)
(* The main program reads in user data and displays the
   tree in Infix, Prefix, and Postfix notation. *)
begin (* TreeTraversal *)
    reset(DataFile,':F1:DATA2');
    writeln; writeln; writeln;
    read(DataFile,NodeCharacter,NodeIndex);
    while not eof(DataFile) do begin
        BuildTree;
        writeln; writeln('INFIX:');
        Infix(One);
        writeln; writeln('PREFIX:');
        Prefix(One);
        writeln; writeln('POSTFIX:');
        Postfix(One);
        writeln; writeln
        end;
    writeln; writeln
end. (* TreeTraversal *)
```

Figure 2-1. Sample Program 2A: Binary Tree Traversal in Standard Pascal (Cont'd.)

The *program block* is the outer-level block in a program. The procedure and function declarations contained within the program block are also blocks. In standard Pascal, procedure and function blocks cannot stand alone; however, they may contain all three parts of a block, including other procedure and function declarations.

Figure 2-1 consists of a program block containing four second-level procedure declarations, marked BLOCK 2, BLOCK 3, BLOCK 4, and BLOCK 5. BLOCK 2 itself contains another procedure declaration, marked BLOCK 6. Thus the structure is hierarchical.

This block structure encourages top-down development and stepwise-refinement techniques of program design. The nesting level of a procedure or function may correspond to the level of that operation in a stepwise breakdown of the problem. For example, for the tree traversal program of figure 2-1, the programmer first divided the task into four main parts: build tree from user input, write out tree in infix notation, write out tree in prefix notation, and write out tree in postfix notation. The program may perform each of these tasks more than once. Next, within the "build tree from user input" task, he identified a sub-task: add a node to the tree. Larger programs may include many more levels of nesting.

The program block defines the main program, with which execution starts when you run the program. During execution, the outer statement part of the program block may make calls to subprogram blocks (procedures and functions) contained within the program block. Thus, the statement part specifies the order in which the program performs the sub-tasks.

Procedures and functions are similar structurally, but differ in how they are invoked and in their purpose. A *procedure* is invoked via a procedure statement (similar to a "call" in some other languages). A *function* is invoked by giving its name and a list of arguments in an expression within a statement. The function returns a value to the calling program; this value replaces the function name and argument list in the expression. Thus procedures often perform actions that change many values, but the primary purpose of a function is to return a single value.

For example, the following function returns the absolute value of the argument corresponding to x:

```
function abs (x:real): real;
const zero = 0.0;
begin
 if x>=zero then abs := x
 else abs := -x
end
```

A *reentrant* procedure or function is one that can be invoked again before the first invocation is finished. This may occur, for example, if an interrupt occurs while the procedure is executing. In Pascal, all procedures and functions are automatically reentrant, as long as they do not change any global variables (defined in 4.1.2). Thus you do not need to make any special provisions for a procedure which may be interrupted during its execution, and subsequently invoked to process the interrupt.

Another important capability of Pascal is recursion. A *recursive* procedure or function is one that calls itself—either directly or indirectly. For instance, if procedure A contains a call to procedure A, it is (directly) recursive. If A calls B and B calls A, the two procedures are both indirectly (mutually) recursive. If A calls B, B calls C, C calls D, and D calls A, all four procedures are indirectly (mutually) recursive. A recursive procedure or function is frequently a natural way to express an algorithm. In figure 2-1, the procedures Infix, Prefix, and Postfix (BLOCK 3, BLOCK 4, and BLOCK 5) are directly recursive.

Certain useful procedures and functions are included as part of the Pascal-86 language. These include arithmetic and conversion functions, ordinal and Boolean functions, input/output procedures, procedures for allocating dynamic variables, and procedures and functions to communicate directly with microprocessor hardware. Some of these are from standard Pascal; others are Intel extensions. You need not declare these *predefined procedures* and *predefined functions* in your program; simply invoke them, using procedure statements or expressions, when needed. The program of figure 2-1 calls the predefined procedures **read**, **readln**, **write**, and **writeln**.

2.2 Separately Compiled Modules

In Pascal-86, you may collect logically related constants, types, variables, procedures, and functions into separately compiled *modules*. Modules are useful for several purposes:

- To allow you to divide a large program into smaller, more manageable parts
- To permit several programmers to work on the same program
- To enable you to locate different parts of your code in different types of memory, or in memory located on different parts of your application hardware
- To allow you to link new code to existing compiled, debugged routines, including general-purpose library routines
- To allow you to code some of your routines in Pascal-86 and some in other languages, such as ASM86 or PL/M-86
- To enable you to divide your code into overlay segments

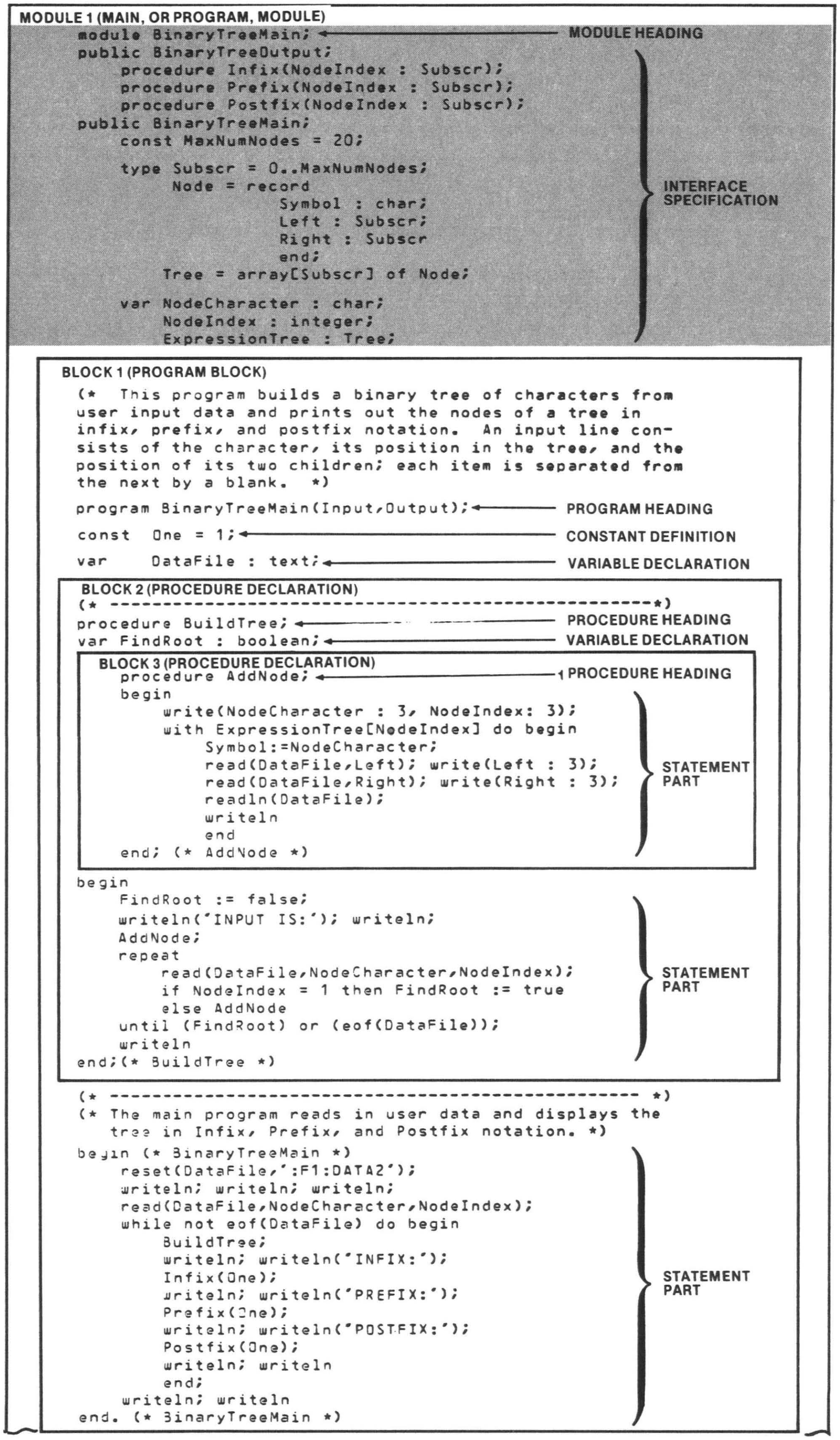
Figure 2-2 shows how you might subdivide the binary tree traversal program into two separately compiled modules. The first module contains the outer-level statements of the program block, and is therefore called the *program module*. It also contains the procedure **BuildTree** and its embedded procedure **AddNode**. The second module contains the procedures **Infix**, **Prefix**, and **Postfix**. Note that these procedures are no longer physically contained within the program block.

A special section of code, called the *interface specification*, is required at the beginning of each separately compiled module to establish communication between the modules. The interface specification indicates which labels, constants, types, variables, procedures, and functions in other modules are accessible to the given module, and which such objects in the given module are accessible to other modules. The use of interface specifications is discussed more fully in 4.2.2.

Program Segmentation

To make a large program more manageable, it can either be broken into separately compiled modules or into a group of subsystems. Separately compiled modules obey one of the simple program segmentation controls (SMALL, COMPACT, or LARGE) discussed in Chapter 10. (Each program module must be compiled with the same segmentation control.) You select a particular segmentation control depending on the total amount of code and data storage required for your program.

Subsystems are also made up of modules compiled under one of the simple program segmentation controls. All the modules in a given subsystem must be compiled with the same control, though the subsystems that make up a program may be of different segmentation schemes. Using subsystems, you can take full advantage of the 8086's segmented architecture, which minimizes the storage needed for pointers, and reduces the number of long procedure calls in a program. A full discussion of subsystems appears in Appendix I.



```
module BinaryTreeMain;
public BinaryTreeOutput;
    procedure Infix(NodeIndex : Subscr);
    procedure Prefix(NodeIndex : Subscr);
    procedure Postfix(NodeIndex : Subscr);
public BinaryTreeMain;
    const MaxNumNodes = 20;

    type Subscr = 0..MaxNumNodes;
         Node = record
                    Symbol : char;
                    Left : Subscr;
                    Right : Subscr
                    end;
        Tree = array[Subscr] of Node;

    var NodeCharacter : char;
        NodeIndex : integer;
        ExpressionTree : Tree;

(*  This program builds a binary tree of characters from
user input data and prints out the nodes of a tree in
infix, prefix, and postfix notation.  An input line con-
sists of the character, its position in the tree, and the
position of its two children; each item is separated from
the next by a blank.  *)

program BinaryTreeMain(Input,Output);

const  One = 1;

var    DataFile : text;

(* ---------------------------------------------------*)
procedure BuildTree;
var FindRoot : boolean;
    procedure AddNode;
    begin
        write(NodeCharacter : 3, NodeIndex: 3);
        with ExpressionTree[NodeIndex] do begin
            Symbol:=NodeCharacter;
            read(DataFile,Left); write(Left : 3);
            read(DataFile,Right); write(Right : 3);
            readln(DataFile);
            writeln
            end
    end; (* AddNode *)
begin
    FindRoot := false;
    writeln('INPUT IS:'); writeln;
    AddNode;
    repeat
        read(DataFile,NodeCharacter,NodeIndex);
        if NodeIndex = 1 then FindRoot := true
        else AddNode
    until (FindRoot) or (eof(DataFile));
    writeln
end;(* BuildTree *)

(* ------------------------------------------------- *)
(* The main program reads in user data and displays the
   tree in Infix, Prefix, and Postfix notation. *)

begin (* BinaryTreeMain *)
    reset(DataFile,':F1:DATA2');
    writeln; writeln; writeln;
    read(DataFile,NodeCharacter,NodeIndex);
    while not eof(DataFile) do begin
        BuildTree;
        writeln; writeln('INFIX:');
        Infix(One);
        writeln; writeln('PREFIX:');
        Prefix(One);
        writeln; writeln('POSTFIX:');
        Postfix(One);
        writeln; writeln
        end;
    writeln; writeln
end. (* BinaryTreeMain *)
```

Figure 2-2. Sample Program 2B: Binary Tree Traversal Using Separately Compiled Modules

MODULE 2 (NON-MAIN MODULE)

```
module BinaryTreeOutput;
public BinaryTreeOutput;
    procedure Infix(NodeIndex : Subscr);
    procedure Prefix(NodeIndex : Subscr);
    procedure Postfix(NodeIndex : Subscr);
public BinaryTreeMain;
    const MaxNumNodes = 20;

    type Subscr = 0..MaxNumNodes;
         Node = record
                   Symbol : char;
                   Left : Subscr;
                   Right : Subscr
                   end;
         Tree = array[Subscr] of Node;

    var NodeCharacter : char;
        NodeIndex : integer;
        ExpressionTree : Tree;

private BinaryTreeOutput;
```

MODULE HEADING

INTERFACE SPECIFICATION

BLOCK 4 (PROCEDURE DECLARATION)

```
(* -------------------------------------------------------- *)
procedure Infix(NodeIndex : Subscr); (* write out the
       tree in infix notation *)
begin
    with ExpressionTree[NodeIndex] do
        if Left <> 0 then begin
            write('(' : 1);
            Infix(Left);
            write(Symbol : 2);
            Infix(Right);
            write(')' : 1)
            end (* if *)
        else write(Symbol : 2)
end; (* Infix *)
```

PROCEDURE HEADING

STATEMENT PART

BLOCK 5 (PROCEDURE DECLARATION)

```
(* -------------------------------------------------------- *)
procedure Prefix(NodeIndex : Subscr); (* write out the
       tree in prefix notation *)
begin
    with ExpressionTree[NodeIndex] do
        if Left <> 0 then begin
            write(Symbol : 2);
            Prefix(Left);
            Prefix(Right)
            end   (* if *)
        else write(Symbol : 2)
end; (* Prefix *)
```

PROCEDURE HEADING

STATEMENT PART

BLOCK 6 (PROCEDURE DECLARATION)

```
(* -------------------------------------------------------- *)
procedure Postfix(NodeIndex : Subscr); (* write out the
       tree in postfix notation *)
begin
    with ExpressionTree[NodeIndex] do
        if Left <> 0 then begin
            Postfix(Left);
            Postfix(Right);
            write(Symbol : 2)
            end (* if *)
        else write(Symbol : 2)
end; (* Postfix *).
```

PROCEDURE HEADING

STATEMENT PART

Figure 2-2. Sample Program 2B: Binary Tree Traversal Using Separately Compiled Modules (Cont'd.)

CHAPTER 3
LANGUAGE ELEMENTS

Chapter 2 presented the overall structure of a Pascal-86 program. This chapter, and the chapters following it through Chapter 8, define the details of the Pascal-86 language.

Rules governing the coding of programs are of two types: syntax rules and semantic rules. The *syntax* of a programming language is the set of rules defining what sequences of symbols make up acceptable programs in the language. The *semantics* of a language is the set of rules for determining, given a syntactically acceptable program, what that program means—that is, what actions it will cause the processor to take. It is possible for a program to be syntactically correct but semantically meaningless.

In this manual, the syntax of each part of a program is generally defined using the syntax notation described in the preface (How To Use This Manual). In cases where such symbolic definitions would be cumbersome, the syntax is presented in prose. Most of these cases occur in this chapter.

Along with the syntax definition of each language construct, the accompanying semantic rules are given in prose. The syntax and the semantic rules are generally followed by one or more examples of the language construct being defined.

The syntax of the entire Pascal-86 language, in syntax notation, is given in Appendix D. For those who prefer the syntax diagrams used in an appendix to the *Pascal User Manual and Report* by Kathleen Jensen and Nicklaus Wirth, and in a number of textbooks on Pascal, Appendix E provides the syntax of the language in that form.

This chapter defines the symbols that make up the building blocks, or "words," of a program. These elements, such as digits, blanks, keywords, identifiers, and special punctuation symbols, make up the larger "sentences" of the language as defined in the chapters that follow.

3.1 Basic Alphabet

The basic building blocks of a Pascal-86 program are:

- The upper-case and lower-case letters
- The digits 0 through 9
- The following keywords (word symbols):

AND
ARRAY
BEGIN
CASE
CONST
DIV
DO
DOWNTO
ELSE
END
FILE
FOR
FUNCTION
GOTO
IF
IN
LABEL
MOD
MODULE
NIL
NOT
OF
OR
OTHERWISE
PACKED
PRIVATE
PROCEDURE
PROGRAM
PUBLIC
RECORD
REPEAT
SET
THEN
TO
TYPE
UNTIL
VAR
WHILE
WITH

- The following punctuation symbols:

+	plus sign
—	minus sign
★	star
/	slash
=	equal sign
<>	"not equal" symbol
<	"less than" symbol
>	"greater than" symbol
<=	"less than or equal to" symbol
>=	"greater than or equal to" symbol
(	left parenthesis
)	right parenthesis
[	left bracket
]	right bracket
{	left brace
}	right brace
:=	assignment symbol
.	period, or dot
,	comma
;	semicolon
:	colon
'	apostrophe
↑	up-arrow
..	ellipsis
_	underscore
@	"at" sign

The keywords are reserved words; that is, you cannot use them as identifiers. A complete list of key words appears in Appendix F. In the syntax notation and in the syntax diagrams, keywords are written in all capital letters. However, you need not capitalize them in your programs.

In keywords and identifiers, a lower-case letter is equivalent to its upper-case counterpart, unless the letter is within a literal character string as defined in 3.3.5. For example:

```
BEGIN    begin    Begin
```

are all the same Pascal-86 keyword. Likewise:

```
TIMEOUT    timeout    TimeOut
```

are all the same identifier.

The use of most of the keywords and punctuation symbols is defined in Chapters 2, 4, 5, 6, and 7. However, here are a few notes about two of the punctuation symbols.

First, a period, or dot (.), marks the end of every compilation (main program module or non-main module).

Second, the semicolon (;) serves as a *separator* between phrases of the language in a program. This differs from the way the semicolon is used in some other high-level languages, including PL/M and PL/1. In these other languages the semicolon is a statement *terminator*; that is, it marks the end of every statement. In Pascal, there is no such terminator. To illustrate the distinction, notice that in a PL/1 program, every statement in a BEGIN block ends with a semicolon:

```
BEGIN;
   TEMP = A;
   A = B;
   B = TEMP;
END;
```

whereas in a Pascal compound statement, the semicolon appears only between the component statements:

```
BEGIN
    TEMP := A;
    A := B;
    B := TEMP
END
```

and the punctuation after END depends upon the context.

Note that in the IF statement defined in 7.2.4, the ELSE clause (if included) is a part of the statement, so you may *not* precede the ELSE clause with a semicolon. The syntax notation and examples in Chapters 4, 5, 6, and 7 make clear when you need to use the semicolon separator.

3.2 Logical Blanks

Declarations, statements, and other constructs in Pascal are in free format; in other words, they are separated from one another by appropriate punctuation or blanks rather than by their positions in the input line. Thus you may extend a declaration or statement over several input lines and indent it for maximum readability and understandability by inserting carriage returns and blanks. Continuation line markers (such as those used in FORTRAN programs) are not needed in Pascal.

Logical blanks are blank characters or blank substitutes that may separate symbols in a Pascal-86 program. The entities that can substitute for a blank are the carriage return character (CR), the line feed character (LF), the tab character (HT), and the comment. The following rules govern the use of logical blanks:

1. Wherever a logical blank is permissible, a sequence of logical blanks is permissible.
2. If a symbol ends in a letter or digit, and the symbol immediately following it begins with a letter or digit, at least one logical blank is *required* to separate them.
3. Embedded logical blanks are not permitted within a keyword, punctuation symbol, identifier, integer, or real number. If such embedded logical blanks do appear, the separated parts become two distinct symbols.
4. One or more logical blanks are permitted, but not required, between any pair of symbols not fitting the cases covered in rules 2 and 3.

3.2.1 Comments

A *comment* in Pascal is a sequence of characters enclosed between a left and a right comment bracketing symbol (and including those symbols). The compiler ignores comments in translating your source program into object code (except that it treats them as logical blanks), but it copies all comments verbatim into the print file along with the rest of your source program. Thus they provide a means for you to insert explanations into your program.

The left comment bracketing symbols are { and (★. The right comment bracketing symbols are } and ★). Either right bracketing symbol may match either left bracketing symbol.

Because the carriage return and line feed are part of the ASCII character set, comments are not limited to a single line. The following are all legal Pascal-86 comments:

```
{this is a comment}

(* this is also a comment *)

(*this is
a comment,
too}
```

3.3 Tokens

From section 3.2, it follows that a Pascal-86 source program compilation consists of a sequence of symbols called *tokens*, or entities indivisible by logical blanks, which may be separated from each other by logical blanks. A token in Pascal is a keyword or punctuation symbol as defined in 3.1, an identifier, an integer, a real number, or a string.

3.3.1 Identifiers

Identifiers in a Pascal program are names used to denote modules, procedures, functions, constants, types, variables, parameters, and field designators. An identifier is a sequence of letters, underscores, and/or digits, of which the first must be a letter. An identifier may be up to 255 characters long. All characters are significant in distinguishing between identifiers.

For example, the following are all legal identifiers:

```
DirectorySearch   COLOR   pi   CAR54
```

You define identifiers in module and program headings, procedure and function declarations, constant definitions, type definitions, and variable declarations. In addition, certain *predefined identifiers* are part of the Pascal-86 language. These stand for predefined procedures and functions, predefined constants, and predefined types that you may use without defining them explicitly. Examples of predefined identifiers are INTEGER, REAL, MAXINT, ABS, READLN, and TEXT. Appendix F gives a complete list.

The association of an identifier with the object it represents must be unique within the scope of the definition or declaration. (Section 4.1.2 discusses scope.) For instance, if you define the identifier INCREMENT as the constant 1.0 in the outer level of a given procedure, you cannot later declare INCREMENT a REAL variable in the outer level of the same procedure.

The Pascal-86 keywords listed in section 3.1 are reserved words, i.e., you may not use them as identifiers. However, the directive FORWARD and the predefined identifiers are not reserved words, so you may use these names. If you use a predefined identifier in a declaration or definition, the effect is to redefine that identifier for the scope of your declaration or definition; thus the predefined meaning is not available within that scope. However, you can declare or define FORWARD as an identifier and also use it in its Pascal-defined meaning as a directive as described in 6.5—the context determines which meaning is intended. The scope of a predefined identifier is the entire compilation less any block where it is redefined.

3.3.2 Integers

A literal *integer* is the textual representation of a decimal, binary, octal, or hexadecimal integer. It is therefore a sequence of decimal digits; a sequence of binary digits (0's and 1's) terminated by the letter B (uppercase or lowercase); a sequence of octal digits (digits 0 through 7) terminated by the letter Q; or a sequence of hexadecimal digits (digits 0 through 9 plus letters A through F), of which the left-most digit must be in the set 0 through 9, terminated by the letter H. Note that hexadecimal numbers must begin with a decimal digit.

A *signed integer* denotes a value of an integer type as defined in 5.3.1, preceded by an optional plus or minus sign. The integer part cannot contain embedded logical blanks, but logical blanks are permitted between the sign and the integer part. An *unsigned integer* denotes an integer value that is not preceded by a plus or minus sign.

The following examples are all legal signed integers. Note that the four signed integers in the top row all represent the same value.

```
10      +12Q       1010B       0Ah
1       -490000    65535    -32749
```

3.3.3 Real Numbers

A literal *real number* is the textual representation of a decimal number that includes a fractional part—that is, one or more digits to the right of the decimal point—or a decimal scale factor, or both. A *signed real number* is a real number preceded by an optional plus or a minus sign. Thus a real number has one of these two forms:

[*sign*]*digits* . *digits*[E [*sign*]*digits*]
[*sign*]*digits* E [*sign*]*digits*

where

sign	is a plus or a minus sign.
digits	is a sequence of one or more decimal digits.

As the syntax shows, if a real number contains a decimal point, it must have at least one digit on each side of the decimal point; for example, −1. and .5 are not legal real numbers.

A real number denotes a constant value of a real type as defined in 5.3.1. It is interpreted as a floating-point number in scientific notation, where the E symbol means "times 10 to the power." The value of a real number must lie in the range permitted for the 80-bit extended precision, or TEMPREAL, numbers, as defined in Appendix H.

The signed real number syntax given in this section applies only to real numbers specified literally in programs. Run-time real text file input to the predefined procedures READ and READLN has the less restrictive syntax defined in 8.7.5.

The following are all legal signed real numbers:

```
 0.1  87.35E+         0.456E+308
5E-3     1E4932     +9.231E-1023
-1.0
```

3.3.4 Labels

A *label* is a sequence of decimal digits that marks a statement so that a GOTO statement may refer to it. It looks the same as a decimal integer, and is distinguished from other labels by its integral value. (For instance, the labels 5 and 005 are the same label and cannot be used in the same block.) In standard Pascal, label values must lie in the range 0 to 9999, inclusive; however, in Pascal-86 a label may be any string of decimal digits that fits on a single line.

A label is distinguished from an integer constant by its context. If a sequence of digits appears in a label declaration, in a GOTO statement, or in the label position at the beginning of a statement, it is interpreted as a label; otherwise, it is considered an integer constant.

3.3.5 Character Strings

A literal *string* is a sequence of one or more characters enclosed by apostrophes. Strings consisting of a single enclosed character denote constants of the predefined type CHAR (5.2.1). Strings consisting of n enclosed characters (n>1) denote constants of the type PACKED ARRAY [1..n] OF CHAR (5.2.2). However, a single-character string constant may also be used where a string is called for; in this case, it is interpreted as type PACKED ARRAY [1..1] OF CHAR. A single-character string constant may not be used in the RESET and REWRITE procedures (see 8.7.1 and 8.7.2).

Strings may contain any of the printable characters in the ASCII character set; however, an apostrophe within a string must appear twice. The printable characters are all the characters with code values from 20H to 7FH inclusive, as defined in Appendix G.

A string may be continued across an input record boundary by closing the string at the end of the line with an apostrophe, then reopening it on the next line with another apostrophe. (Logical blanks may appear before the second opening apostrophe.)

The following are all legal strings. The first three are of type CHAR; the last two are of type PACKED ARRAY [1..16] OF CHAR.

```
1. 'A'
2. ':'
3. ''''
4. 'OVERFLOW ERROR 5'
5. 'This is'
   ' a string'
```

CHAPTER 4
PROGRAM HEADINGS, SEPARATE COMPILATION FACILITIES, AND LABEL DECLARATIONS

Using the basic building blocks defined in Chapter 3, which you can think of as "words," you write headings, definitions, declarations, statements, and the other larger constructs, or "sentences," of the Pascal-86 language. This chapter defines the syntax and semantics of the "sentences" appearing at the beginning of a program, module, or block, before the data definitions: program headings, separate compilation facilities (module headings, interface specifications, and private headings), and label declarations.

Before describing these constructs, this chapter provides necessary information on program structure, filling in details not covered in Chapter 2.

4.1 Details of Program Structure

4.1.1 Parts of a Program

Figure 4-1 shows the parts of a standard Pascal program and of a block, in the order in which they must appear.

The program heading gives the module a name. The program heading may also include a program parameter list, which is a list of files used for input and output in the program.

The label declaration defines statement labels used in the statement part. The constant definitions, type definitions, and variable declarations define the data items used. The procedure and function declarations are blocks that define sub-programs, and may themselves include embedded blocks.

The statement part is a compound statement, which consists of one or more embedded statements enclosed between the keywords BEGIN and END. These embedded statements specify the actions to be performed when the block is invoked during execution. The statement part at the outer level contains the statements that are invoked by the operating system to start execution of the program.

The block shown in figure 4-1 is a basic Pascal block. Program blocks and procedure and function blocks consist of the basic block preceded by an appropriate heading and—for a program block—followed by a period.

For an example of program structure, refer to figure 2-1. Note that there are no label declarations (none are needed, since no statements are labeled and there are no GOTO statements). Also note the block nesting in the procedure and function declarations. A block is always recognizable by its statement part; the other parts may be absent.

Main and Non-Main Modules

As discussed in Chapter 2, you may partition your Pascal-86 program into smaller units, called modules, for separate development and compilation. These modules are of two types: the main program module and the non-main modules. Every program must have one and only one main program module; it need not be coded in Pascal. Information on coding main and non-main modules in other languages is discussed in Appendix J.

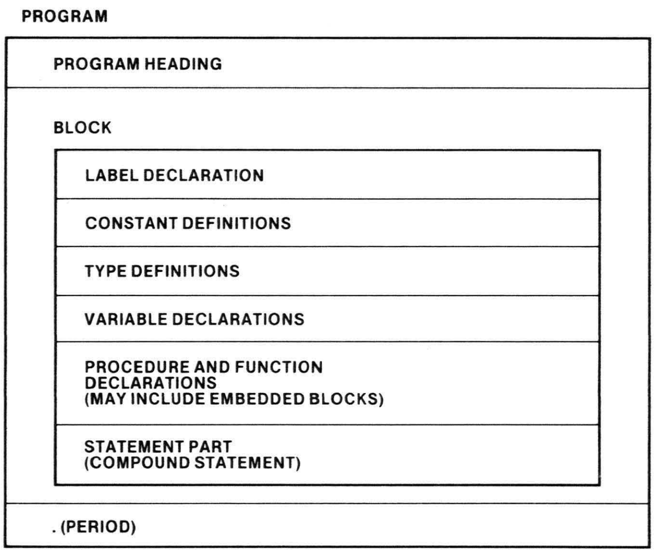


Figure 4-1. Parts of a Standard Pascal Program and Block 121539-30

Figures 4-2 and 4-3 show the parts of a Pascal-86 main program module and a non-main module when the separate compilation facilities are used. Note that the main program module is the same as a standard Pascal main program, with a module heading and an interface specification added at the beginning.

Each compilation, whether it is a main program module or a non-main module, may begin with a module heading followed by an interface specification. Following that is either the program heading (for a main program module) or the private heading (for a non-main module).

Notice the similarities and differences between the main program module and the non-main module. Every module begins with an (optional) module heading and interface specification followed by either a program heading or a private heading, and ends with a period. Between the interface specification and the period, however, the main program module contains a complete block, whereas the non-main module contains a declaration/definition part—which is similar to a block but lacks a label declaration and a statement part. Thus the main program module is the only module in a program that may have a label declaration or a statement part at its outer level.

The module heading identifies the module by name. This name must match the name given in the program heading or private heading. The interface specification provides the only means for modules to communicate with each other. It defines the objects in

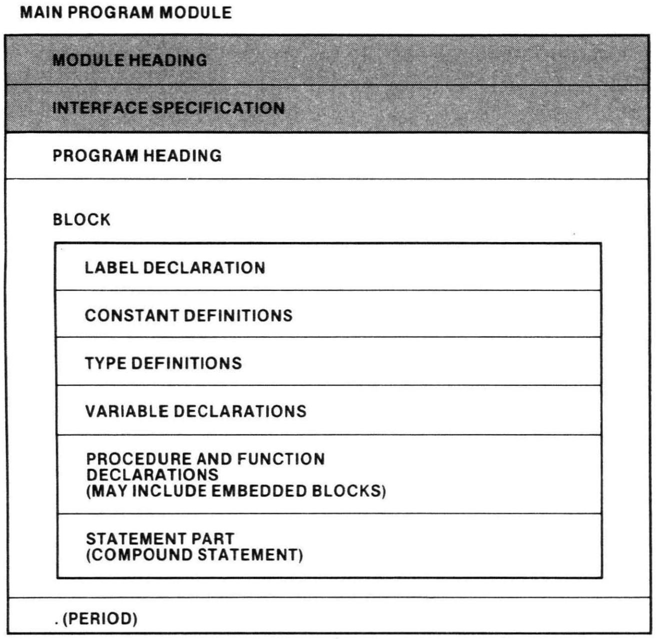


Figure 4-2. Parts of a Pascal-86 Main Program Module 121539-31

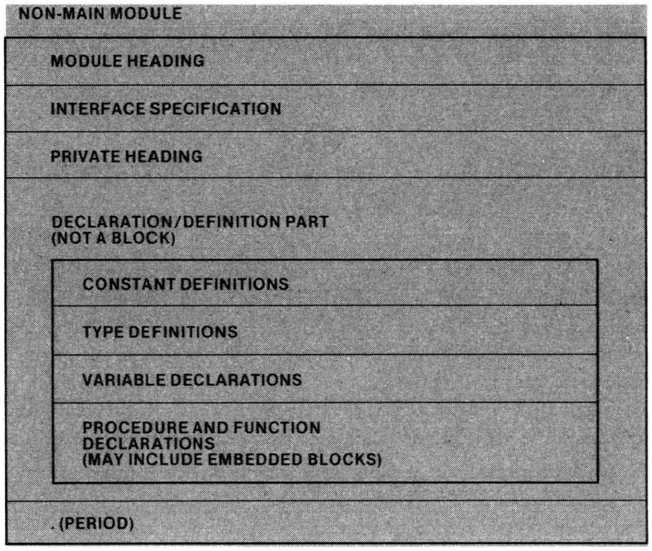


Figure 4-3. Parts of a Pascal-86 Non-Main Module 121539-32

other modules that may be referenced by this module, and also defines the objects in this module that may be referenced by other modules. Both these items are optional, since you do not need them if your program consists of only one module. However, if you include an interface specification, you must include a module heading.

The program heading gives the main program module a name; the private heading names the non-main module. This name must match the name you give in the module heading. Only the program heading may include a program parameter list.

For an example of a separately compiled main module and non-main module, see figure 2-2.

In syntax notation, a main program module has the following form:

```
[module-heading  ;
[interface-spec]
program-heading  ;
block  .
```

and the syntax of a *block* is:

```
[label-decl]
[CONST  constant-defn  ;  [constant-defn  ;]. . .]
[TYPE  type-defn  ;  [type-defn  ;]. . .]
[VAR  variable-decl  ;  [variable-decl  ;]. . .]
[proc-or-func  ;]. . .
statement-part
```

Similarly, the syntax of a *non-main module* is as follows:

```
[module-heading  ;
[interface-spec]
private-heading  ;
declordefn  .
```

where *declordefn* is the declaration-definition part, with the following syntax:

```
[CONST  constant-defn  ;  [constant-defn  ;]. . .]
[TYPE  type-defn  ;  [type-defn  ;]. . .]
[VAR  variable-decl  ;  [variable-decl  ;]. . .]
[proc-or-func  ;]. . .
```

In the syntax definitions, *interface-spec* stands for an interface specification; *defn* and *decl* stand for definition and declaration, respectively, and *proc-or-func* denotes either a procedure declaration or a function declaration. The notation for the latter indicates that procedure declarations and function declarations can be intermixed.

Section 4.2 of this chapter defines module headings, interface specifications, program headings, private headings, and label declarations. Chapter 5 discusses data definitions and declarations (constant definitions, type definitions, and variable declarations). Chapter 6 covers procedure and function declarations, and Chapter 7 defines the statements that can appear in the statement part.

4.1.2 Program Objects and Scope

Objects in a Pascal program include modules, programs, functions, procedures, parameters, constants, types, variables, fields, and labels. You choose appropriate names as symbols for these objects; for instance, the programmer used the name

TreeTraversal for the program of figure 2-1, and likewise chose descriptive names for the constants, types, and variables in the program. Identifiers in Pascal can be as long as a line of text (255 characters), so you can choose names that make your programs easy to understand.

A definition or declaration introduces an object and associates it with, or binds it to, a symbol. This symbol must be an identifier (for most objects) or a decimal integer constant (for a label). The *scope* of a definition or declaration is the part of the source program over which that association holds. Generally, a scope is a block, statement, parameter list, field designator, or other language construct, excluding any enclosed constructs that set up another scope for the same symbol.

Public and Private Objects

The objects of a module, which include labels, constants, types, variables, procedures, and functions, may be either private or public. *Private* objects are those that only that program module may reference. *Public* objects are objects that other designated modules of the program may freely reference.

The *private section* of a module defines the private objects, and the body of public procedures, belonging to that module. The *public section* of a module specifies which objects belonging to that module are public objects. A module always has one, and only one, private section; this is the main body of the module (the part following the program heading or private heading). A module may have more than one public section, but all public sections are optional.

Local and Global Objects

Objects declared or defined at the outer level of a program block, and thus usable by all subprograms, are *global* objects. Object declared or defined within a procedure or function are *local* objects, said to be local to that procedure or function. When several programmers are working on a large program, each programmer is concerned only with the global objects, such as global variables shared by all parts of the program, and with the local objects in the procedures he is writing. Note that all public objects declared in the interface specification (4.2.2) have global scope.

The tree traversal program, for example, defines two constants and three types, and declares four variables and four procedures, at its outer level. These are all global objects. The constants are **MaxNumNodes** and **One**; the types, **Subscr**, **Node**, and **Tree**; the variables, **NodeCharacter**, **NodeIndex**, **ExpressionTree**, and **DataFile**; the procedures, **BuildTree**, **Infix**, **Prefix**, and **Postfix**. The scope of these declarations and definitions is the entire program, excluding the one-line program heading.

On the other hand, the local variable **FindRoot** is declared within the procedure **BuildTree**, so its scope is only the procedure **BuildTree**. This scope includes the statement part of **BuildTree** plus the contained procedure **AddNode**, but does not include any of the other procedures or the outer-level statement part. If the program had also declared a variable named **FindRoot** at its outer level, two variables with that name would exist. The scope of the inner **FindRoot** variable would remain the same, and the scope of the outer **FindRoot** would be the entire program *except* for the scope of the inner **FindRoot**. If another procedure, such as **Infix**, had declared a variable named **FindRoot**, the scope of that variable would be its containing procedure, so there would be no conflict with the other variables of the same name.

Thus if several programmers are working on a large program, they need not care whether they use some of the same names for local variables. All the programmers need to agree upon are the names of global objects.

Only global data exist during the entire execution of a Pascal program; data local to a subprogram are created automatically each time the subprogram is invoked, and disappear when execution returns from the subprogram. This rule enforces program clarity, since the programmer must declare all permanent data in one place in the program—at the outer level. As a side effect, the release of local storage saves memory space.

A program must open files explicitly using the predefined procedure RESET or REWRITE. However, the run-time system closes files automatically at the end of the program.

Defining Identifiers

The program objects represented by predefined identifiers (discussed in 3.3.1) are assumed to be declared or defined at the outer level of every program or module. If you define one of these identifiers for your own purposes, your own declaration or definition, within its own scope only, overrides the predefinition. Thus you need not be concerned with such name conflicts unless you wish to use the predefined items. Also, you may easily replace most predefined procedures, functions, or other objects with your own versions.

All labels and identifiers you use in a program, except predefined identifiers, must be declared or defined for the parts of the program in which you use them. For instance, if a procedure in your program uses the identifiers X and Y as variables in a calculation, you must declare them either within that procedure, within a containing procedure, or at the outer level of the program. This rule helps eliminate invisible side effects in programs, and allows the compiler to perform type and range checking to detect errors.

Parameters

A *parameter list* is a list of identifiers in the heading of a program, procedure, or function. These identifiers denote objects through which the program, procedure, or function communicates with its environment. Parameters define explicitly the nature of the interface between the program or subprogram and its environment. They also allow the compiler to check for certain kinds of programming errors.

In the tree traversal program, the procedures **BuildTree** and **AddNode** have no parameters. The procedures **Infix**, **Prefix**, and **Postfix** each have one parameter: **NodeIndex**, of type **Subscr**.

In a program heading, the parameters denote objects that exist outside the program—that is, files used for input and output. For example, the tree traversal program names the standard files INPUT and OUTPUT as program parameters.

In a procedure or function heading, the parameters likewise refer to objects in the environment outside the procedure or function. However, these parameters match up with corresponding identifiers in an *argument list* in the statement or expression that invokes the procedure or function. Because of this matching, the objects inside the procedure or function do not need to have the same names as the corresponding objects outside. This naming independence is useful when you need to perform the same operation on several different variables in a program.

For instance, if you have written a matrix multiplication procedure with the heading:

```
procedure MatrixMult (Matrix1, Matrix2: MatrixType;
    VAR OutMatrix: MatrixType);
```

and you need to multiply the three matrix pairs A × B, C × D, and E × F (all six matrices, plus the result matrices X, Y, and Z, being of type **MatrixType**, defined as:

```
type MatrixType = array [1..10,1..10] of REAL;
```

you could perform the multiplications with the statements:

```
MatrixMult(A,B,X);
MatrixMult(C,D,Y);
MatrixMult(E,F,Z);
```

4.2 Program Headings and Separate Compilation Facilities

This section gives the syntax and semantics of the parts of a program that establish its identity as a program and its division into separately compiled modules. These parts include the module heading, the interface specification, the program heading, and the private heading.

4.2.1 Module Heading

The *module heading* identifies, by name, the module to be compiled. This name must match the name you give in the program or private heading. The syntax of a *module-heading* is:

MODULE *identifier*

where

identifier is the module name given in the program heading (4.2.3) or private heading (4.2.4).

In the interface specification (4.2.2), the public section with the same name as the module heading defines this module's public objects; they are available for use by other modules. Public sections with different module names define the objects that are external to this module.

4.2.2 Interface Specification

The *interface specification* part of a module contains that module's public section and the public sections of other modules that communicate with it. The interface specification thus defines the public objects in this module that may be referenced by other modules, and also the public objects in other modules that may be referenced as external objects by this module.

The syntax of the *interface-spec* is as follows:

```
PUBLIC identifier ;
section
[PUBLIC identifier ;
section]. . .
```

where

```
    section        is subsection [subsection]. . .
    subsection     is [FOR identifier [, identifier]. . . ;]
                      [label-decl]
                      [CONST constant-defn ; [constant-defn ;]. . .]
                      [TYPE type-defn ; [type-defn ;]. . .]
                      [VAR variable-decl ; [variable-decl ;]. . .]
                      [prochdg-or-funchdg ;]. . .
```

Here, *defn* and *decl* stand for definition and declaration, respectively, and *prochdg-or-funchdg* stands for either a procedure heading or a function heading.

Note from the syntax that the same PUBLIC section is used when declaring one module's public objects as when referencing them as external objects from other modules. Thus, the interface specification is designed to be kept in separate files and included in each compilation as needed by means of the INCLUDE control. This practice also ensures that each module's public section is the same in every compilation.

Public Section

The identifier following the keyword PUBLIC is the name of the module whose public section is being declared. For each module, the public section includes those objects defined by the module that can be referenced as externals by all other modules in the program.

Constants, types, and variables are fully defined in the public section, while public procedures and functions are designated only by their headings (similar to FORWARD definitions). You must define the body of each PUBLIC procedure and function in the private section of the corresponding module. Public procedures and functions may not be nested within other procedures and functions.

When a module is compiled, its public section (if it exists) must be present in the interface specification of the module. Other public sections must be present if the private section of the module references objects defined in them, or if other public sections in the compilation do so. (A public section may reference constants or types defined in another public section. In such cases, it is not necessary that the public section defining the objects precede the public section that uses them.) Additional public sections that are not referenced at all may also be included.

FOR-Clause. The *identifiers* in the optional FOR-clause are the names of other modules that are legal recipients of the definitions and declarations in the section—that is, modules whose public and private sections can reference the objects defined and declared in the section. If the FOR-clause is present in a subsection, the only legal recipients of that subsection's definitions and declarations are the modules named in the FOR-clause (and the module whose public section this is, since a module may always access its own public objects). If no FOR-clause is present, all modules in the compilation are legal recipients.

NOTE

To ensure that files are initialized properly, all global file variables must be accessible to the main program module; that is, declared PUBLIC without a FOR clause or a PUBLIC FOR the main module.

Scope of Public Objects. The names of the objects defined in a public section are called *public symbols*. Each public symbol must be unique among the public symbols of a program. In a given compilation, the scope of a public symbol definition is the composite of all the legal recipient sections, public and private, in the compilation. A public symbol always refers to the same object in every compilation in which it is defined. That object is the one specified in the defining public section.

Any identifier referenced in a public section must be a public identifier that the public section may legally receive.

Public Labels. Your main program module may declare a label PUBLIC, so that other modules can jump back to the main program for error recovery. These labels must be in the statement-part of the outer program block. Non-main modules cannot declare public labels.

For example, you might write the following public section for a module named MOD1:

```
PUBLIC MOD1;
   CONST zero = 0.0;
FOR MOD2, MOD3;
   TYPE name = packed array [1..20] of char;
        state = packed array [1..2] of char;
   VAR i, j, k: integer;
FOR MOD3, MOD4;
   VAR m, n: integer;
FOR MOD2;
   LABEL 110, 120, 130;
   TYPE complex = RECORD re, im: real END;
   VAR x, y, z: real;
   FUNCTION sinh (arg: real): real;
   FUNCTION cosh (arg: real): real;
   PROCEDURE graph (abscissa, ordinate: real);
```

The constant **zero** is accessible to all modules. The types **name** and **state** and the variables **i**, **j**, and **k** are accessible only to MOD1, MOD2, and MOD3. The variables **m** and **n** are accessible only to MOD1, MOD3, and MOD4. The labels 110, 120, and 130, the type **complex**, the variables **x**, **y**, and **z**, the functions **sinh** and **cosh**, and the procedure **graph** are accessible only to MOD1 and MOD2. The private section of MOD1 must give declarations (including the procedure or function body) for **sinh**, **cosh**, and **graph**.

For a complete sample program including interface specifications, see figure 2-2 at the end of Chapter 2.

4.2.3 Program Heading

The *program heading* gives a name to a main program module and introduces the private section of that module. The syntax of a *program-heading* is:

PROGRAM *identifier* [(*prog-parameter-list*)]

where

identifier is the name given to the main program module. It must match the name given in the module heading, if there is one.

prog-parameter-list is a list of program parameters, separated by commas, that are names of external objects used by the program. These names should be the names of file variables; objects of any other type cause the compiler to generate a warning.

Pascal predefines the standard program parameters INPUT and OUTPUT, which are text file variables as defined in 5.3.2 and 8.7. If no file argument is given in calls to the predefined procedures and functions READ, READLN, GET, EOF, and EOLN, they assume the file INPUT. Likewise, the procedures WRITE, WRITELN, PUT, and PAGE assume the file OUTPUT.

You must not declare these files as variables in your program, but you must list them as program parameters if you use them in your program. The appearance of these files as program parameters causes them to be declared implicitly in the program block, and the initializing statements RESET (INPUT) and REWRITE (OUTPUT) are automatically generated if required.

You may reference the standard files INPUT and OUTPUT in a non-main module. Both implicit references (using READ, READLN, WRITE, WRITELN, PUT, GET, PAGE, EOF, or EOLN without specifying a file variable) and explicit references to undeclared files named INPUT and OUTPUT are valid. However, if you make such references, you must name these files as program parameters in the program heading of the main program module.

4.2.4 Private Heading

The *private heading* gives a name to a non-main module and introduces the private section of that module. The syntax of the *private heading* is:

PRIVATE *identifier*

where

identifier is the name to be given to the module. This name must match the name given in the module statement.

4.3 Label Declaration

A *label declaration* specifies the integers you use to label statements in the statement part of a block. The syntax of a *label declaration* is:

LABEL *label* [, *label*]. . . ;

where

label is an integer label as described in 3.3.4.

An example is:

```
LABEL 20, 40, 110
```

Each label listed in the declaration must appear in the label position of exactly one statement in the statement part. The scope of a label declaration is the block in which the declaration occurs.

Within the scope of the declaration, integers are interpreted as references to a label only when they appear in GOTO statements and in the label position of a statement. In all other cases, they are interpreted as signed integers.

You need labels and label declarations only when you use GOTO statements in your program. It is considered good programming practice either to avoid using GOTO statements altogether, or to use them only when there is a very good reason for doing so (such as program clarity).

Your main program may declare a label PUBLIC, so that other modules can jump back to the main program for error recovery. These labels must be defined in the statement part of the outer program block. Non-main modules cannot define public labels.

CHAPTER 5
CONSTANTS, TYPES, AND VARIABLES

An algorithm or computer program consists of two parts: a description of the data, and a description of the actions to be performed on it. To describe the data in a Pascal program, you write declarations and definitions; to describe the actions, you write statements. This chapter describes how to define your data—using constants, types, and variables—and denote it in statements. Chapter 7 describes how to write Pascal statements and the expressions within them.

5.1 Basic Concepts

Constants are data items whose values cannot change during execution of a program; *variables* are data items whose values can change, and which the program processes. Every constant and variable in a Pascal program has a type.

The type is a central concept in Pascal. A *type* denotes a set of values which a data item can assume; any definition, declaration, or program operation that requires the data item to assume a value not in that set causes an error. Examples of data types used in other programming languages are INTEGER and REAL; these are also types in Pascal. However, Pascal offers a richer variety of data types than most other languages, and even allows you to define your own.

The type of a *variable* may be either directly described in the variable declaration, or referenced by a type identifier. In the latter case, the identifier must first be described by a type definition. The compatibility of variables is based primarily on the types associated with them.

Variables in Pascal can be generated statically, automatically, or dynamically. Global variables (those declared at the outer level of a module) are *static*; that is, they exist for the entire program run. Variables that you declare explicitly within a procedure or function are *automatic*; that is, they are generated at run time in accordance with the structure of the program. For example, if you have declared a certain variable local to a procedure in your program, one instance of that variable is created whenever the procedure is called, and that instance of the variable is destroyed upon return from that invocation. When you declare a static or automatic variable in your program you give the variable a name, by which you can reference it in the statements of your program, and a type. The block level at which you place that declaration in the program determines the scope of the variable, and thus determines when it is created and destroyed.

For example, in the program of figure 2-1, the global (static) variables **NodeCharacter**, **NodeIndex**, **ExpressionTree**, and **DataFile** exist for the entire program run. In contrast, an instance of the local (automatic) variable **FindRoot** is created each time the **BuildTree** procedure is called, and this instance of the variable disappears when control returns from the invocation of the procedure.

Dynamic variables, on the other hand, are generated by statements within your program, without regard to program structure. You do not declare them explicitly. You create dynamic variables using the predefined procedure NEW, and destroy them using the procedure DISPOSE. Whenever you call NEW to create a dynamic variable, it assigns a value to a variable of a *pointer* type, which you can then use to reference the dynamic variable. Although a dynamic variable is not declared, it still has a type, which is determined by the type you declared for the pointer variable.

Dynamic variables are useful in creating complex data structures, such as linked lists and trees, that must change in form or size as the program runs.

The values of Pascal variables are initially undefined, so you must explicitly initialize all variables in your program. This rule forces you to make clear exactly what initial values you are assuming.

You may represent constants either literally, by their values (e.g., 3.14159, *string*, TRUE), or by symbolic names (e.g., PI). In the latter case, you must define the name and value of each constant in a constant definition. (The type of the constant is implied by its value.)

The passing of data to procedures and functions by means of *parameters* is subject to special rules. Parameters are objects that differ from the types, constants, and variables described in this chapter. As is true for constants and variables, every parameter has a type. Parameters are discussed in detail in Chapter 6.

The remainder of this chapter first discusses constant definitions and the various kinds of data types available in Pascal-86, giving the syntax and semantic rules for defining them and examples of their use with static variables. Following this, the discussion turns to pointer types and dynamic variables. The final sections of the chapter define the form of variable declarations and denotations.

The major headings in the remainder of this chapter (Constants, Types, and Variables) correspond to the order in which you define or declare these objects in your programs. Standard Pascal requires that you define or declare each object before you use it in another definition or declaration. (In a pointer type definition, however, you may make a forward reference to the base type of the pointer.) Pascal-86 does not make this requirement.

5.2 Constants

As mentioned earlier, *constants* are data items whose values cannot change during execution of a program. You may represent constants either literally (as integers, real numbers, or character strings as defined in Chapter 3) or as *named constants*.

To use a named constant, you must first define it in a *constant definition*, which introduces an identifier as a synonym for a constant value (or for another named constant). Then you may use the constant identifier freely in place of its literal value in expressions and in any other places where a constant of its type is permitted.

The constant definitions in a Pascal block or non-main module appear in a list following the keyword CONST, between the label declarations and the type definitions, as indicated in 4.1.1. The syntax of a constant definition (*constant-defn*) is as follows:

identifier = *constant*

where

identifier	is unique.
constant	is an integer (3.3.2), a real number (3.3.3), a character string (3.3.5), a constant identifier, or a numeric constant identifier.

The constant identifier may have been defined either in a constant definition or in the definition of an enumerated type (5.3.1). The type of the constant identifier is the type of the given *constant*.

Note that a real constant (3.3.3) is always represented in TEMPREAL precision, as defined in 5.3.1. (See Appendix H for the internal format of TEMPREAL numbers.)

In Pascal-86, you may make indexed references to individual characters in a named string constant as if it were an array variable (5.4.2).

The following are all legal constant definitions (to be preceded by the keyword CONST and separated by a semicolon if there are several, as indicated in 4.1.1):

```
ScaleFactor = 12
Gamma = 0.577216
EulersConstant = Gamma
filler = '******'
```

Named constants allow you to write programs that are more meaningful and easier to modify. For instance, if you use a constant scaling factor of 12 in a number of places in your program, you can first define the identifier **ScaleFactor** as the value 12 as shown in the first example above; then write **ScaleFactor**, rather than 12, everywhere in the program that the scale factor is needed. Someone reading the program can tell at once, from the name, what the constant means. If you later wish to change the value of the scale factor, you need make only one change in the program—just change the constant definition.

5.3 Types

Pascal is a *strongly typed* language. This means that every data item has a type, and that you must follow strict rules in the use of types in definitions, declarations, and expressions. The compatibility of variables depends primarily on the type associated with them; violation of a type compatibility rule causes an error. Strong typing enables the compiler to do extensive type and range checking, so that you can catch many program errors earlier in the development process—at compile time rather than at run time. Types in Pascal also allow you to phrase your program in meaningful terms—terms related to the problem you are solving.

Figure 5-1 shows the relationship between the various kinds of types in Pascal-86. *Simple types*, called *scalar types* in standard Pascal, are types whose variables have a single value. *Structured types* are types whose variables are made up of a number of single values; these structured types are built up from simple types. *Pointer types* are types whose variables you use to access dynamic variables in your program.

The type definitions in a Pascal block or non-main module appear in a list, separated by semi-colons, following the keyword TYPE, as defined in 4.1.1.

A *type definition* associates a name (identifier) with a set of values. The syntax of a *type-defn* is:

identifier = *type-spec*

where

identifier is unique—that is, it is not defined for any other purpose in that block.

type-spec a type specification which identifies a set of values, has a form that depends on the type being defined. The scope of a type definition is the block in which the definition falls, excluding the parameter list associated with the block.

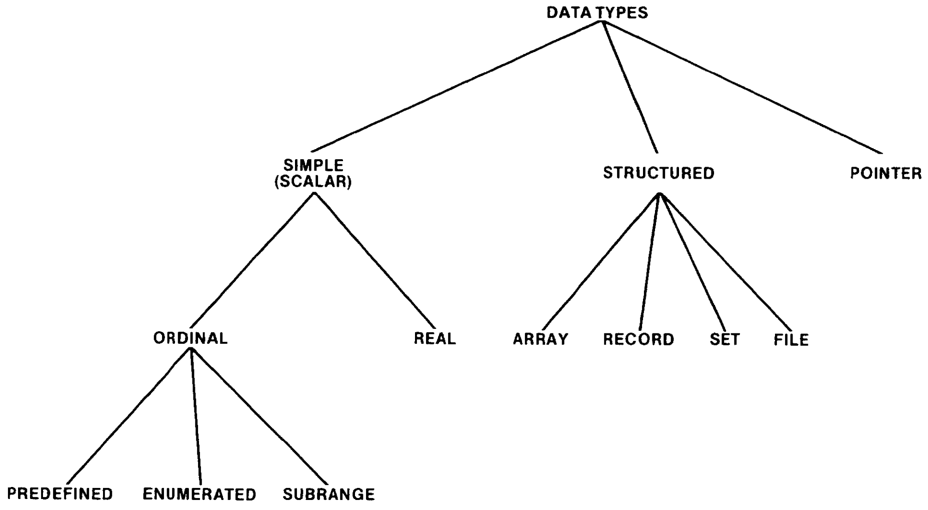


Figure 5-1. Data Types in Pascal 121539-33

```
TYPE counter = INTEGER;
     color = (red, yellow, blue, green, orange, violet);
     colour = color;
     shade = color;
     primarycolor = red .. blue;
     Score = 1..100;
     alfa = PACKED ARRAY [1..80] OF CHAR;
     complex = RECORD re,im: REAL END;
     person = RECORD name, firstname: alfa;
                     age: integer;
                     married: Boolean;
                     father, child, sibling: ↑person;
                     END;
     alphabet = SET OF CHAR;
     characters = FILE OF CHAR;
     manuscript = TEXT;
     Link = ↑complex;
```

In the examples just given, the type **counter** is the same as the predefined simple type INTEGER. The types **color**, **colour**, and **shade** are enumerated types. Type **alfa** is an array type; **complex** and **person** are record types; **alphabet** is a set type; and **characters** and **manuscript** are file types. (TEXT is a predefined file type.) Type **Link** is a pointer type.

As shown in these examples, you may define a type T2 in terms of another type T1. In such cases, in standard Pascal you must always define T1 before using it to define T2. (The Pascal-86 compiler, however, does not check for this violation.) You may not use a recursive type definition—that is, define a type in terms of itself—*except* in a pointer type specification nested within a structured type specification, as shown in the type definition for **person**. When ↑T is used as a *type-spec* you may define T anywhere within the type definition part of that block or the type definition part of an enclosing block.

5.3.1 Simple Types

A constant, variable, or parameter of a simple type consists of a single value. The type specifies the set of values to which that value must belong. For instance, the predefined type INTEGER includes all integers that lie within the representable integer range (in Pascal-86, −32767 through +32767). Thus 5, −20000, and 999 are acceptable values for INTEGER variables, but −40000, 3.6, and **blue** are not.

The set of values denoted by a simple type always has an order, so that one may compare two values of the same type and determine whether the first is greater than, equal to, or less than the second.

A simple type is either an ordinal type or a real type.

Ordinal Types

An *ordinal type* is a simple type whose values can be assigned sequence numbers—that is, mapped onto the set of whole numbers. All types denoting a finite set of values, such as (red, yellow, blue) or (FALSE, TRUE), are ordinal, since you can count the values. The predefined Pascal-86 types INTEGER, WORD, and LONGINT are also ordinal.

The predefined functions ORD, LORD, WRD, CHR, PRED, and SUCC operate on the mapping between ordinal types and the whole numbers. ORD, LORD, and WRD take an expression of ordinal type and return a value of type INTEGER, LONGINT, or WORD, respectively. CHR takes an integer expression and returns the corresponding value of type CHAR (defined in the next section). PRED and SUCC take an expression of ordinal type and return the value of the type that precedes or succeeds it, respectively, in the ordering. Complete descriptions of these functions are given in 8.1.

Predefined Ordinal Types. Three ordinal types are predefined in standard Pascal. These types are denoted, respectively, by the type identifiers INTEGER, BOOLEAN, and CHAR. In addition, Pascal-86 includes three more predefined types, denoted by the identifiers WORD, LONGINT, and AT87EXCEPTIONS (a special type, discussed in 8.10).

The type definition for a predefined ordinal type is specified at the outermost level of the program. You operate on values of a predefined ordinal type by using operators such as addition, comparison, and the Boolean AND, and by invoking predefined procedures and functions.

INTEGER is an ordinal type whose values are a subset of the whole numbers. In Pascal-86, these values are two bytes long and lie in the range −32767 through +32767. INTEGER values are denoted by signed integers (3.3.2) whose values fall within the defined subset. The predefined constant MAXINT specifies the upper bound of the INTEGER range. (In Pascal-86, MAXINT has a value of 32767.)

WORD is an ordinal type whose values are a subset of the whole numbers. In Pascal-86, these values are two bytes long and lie in the range 0 through 65535. WORD values are denoted by unsigned integers (3.3.2) whose values fall within the defined subset. The predefined constant MAXWORD specifies the upper bound of the WORD range. (In Pascal-86, MAXWORD has a value of 65535.)

LONGINT is an ordinal type whose values are a subset of the whole numbers. In Pascal-86, these values are four bytes long and lie in the range −2,147,483,647 through +2,147,483,647. LONGINT values are denoted by signed integers (3.3.2) whose values fall within the defined subset. The predefined constant MAXLONGINT specifies the upper bound of the LONGINT range. (In Pascal-86, MAXLONGINT has a value of 2,147,483,647.)

Note that LONGINT is intended primarily for arithmetic operations; its use in type definitions is relatively restricted. LONGINT cannot be used as the index type of an array, the base type of a set, the tag type of a variant record, or the control variable of a FOR loop. However, since LONGINT is compatible with INTEGER and WORD, a LONGINT expression can appear anywhere that an INTEGER or WORD expression can appear. Although an array cannot be defined to have a LONGINT index type, a LONGINT expression can be used as the index expression in an array reference, provided that the value of the expression is in the range specified by the index type of the array.

BOOLEAN is an ordinal type whose values are the truth values denoted by the predefined identifiers TRUE and FALSE, where FALSE precedes TRUE. Boolean operators are defined on values of type BOOLEAN, and the results of relational operators are always of type BOOLEAN.

CHAR is an ordinal type whose values are a defined set of characters. In Pascal-86, this set is the ASCII character set. They are denoted by the characters themselves enclosed within apostrophes or by the predefined constant identifiers CR and LF, which stand for the ASCII characters carriage return and line feed, respectively. Note that values of type CHAR always consist of a single character; apostrophe-enclosed character strings of more than one character are of type PACKED ARRAY [1..n] OF CHAR, as discussed in 5.3.2.

The ordering properties of the character values are defined by the ordering properties (ordinal values) of the characters in the character set. In other words, the relationship between character values c1 and c2 is the same as the relationship between ORD(c1) and ORD(c2), as defined in 8.1.1. In all Pascal implementations, the following relations hold:

- The subset of character values representing the digits 0 to 9 is ordered numerically and is contiguous.
- The subset of character values representing the upper-case letters A to Z, if available, is ordered alphabetically, but is not necessarily contiguous.
- The subset of character values representing the lower-case letters a to z, if available, is ordered alphabetically, but is not necessarily contiguous.

Appendix G gives the ordering of the character set defined for Pascal-86 (the ASCII set).

Enumerated Types. An *enumerated type* is an ordinal type you define yourself by specifying a list of items; for instance, a list of colors. These items, represented by identifiers, are the set of values that variables of the enumerated type can assume. For instance, if you define a type called **primarycolor** consisting of **red**, **yellow**, and **blue**, and then define a variable of that type called **wallcolor**, the only permissible values for **wallcolor** would be **red**, **yellow**, and **blue**.

The ordering of the values of an enumerated type is determined by the order in which the identifiers are named in the *type-spec*, which has the syntax:

(*identifier* [, *identifier*] . . .)

All the *identifiers* in the list are unique—that is, none appears in the list more than once, and none is defined for any other purpose in that block. Naming these identifiers in the enumerated *type-spec* automatically defines them as constants of that type.

Examples of enumerated *type-specs*:

```
(red, yellow, blue, green, orange, violet)
(club, diamond, heart, spade)
(Monday, Tuesday, Wednesday, Thursday, Friday,
 Saturday, Sunday)
```

Enumerated types allow you to call the items you are dealing with by meaningful names, instead of having to use objects of a more general type (such as integers or Boolean variables) to represent them. Enumerated types can be extremely helpful to you in writing readable, understandable programs.

Also note that the built-in ordering in enumerated types allows you to compare values and variables of these types. For instance, if you define a type SUIT using the second type specification given above, a variable of type SUIT whose value is **spade** will be greater than a variable whose value is **club**.

The Pascal-86 predefined type AT87EXCEPTIONS, discussed in 8.10, is an enumerated type.

Ordinal type transfer functions are described in 8.1.7.

Subrange Types. A *subrange type* is a type you define as a subrange of an ordinal type, called the *host type*. A subrange type denotes a consecutive subset of the values of a previously defined type, such as the integers 1 through 100, the integers −10 through +10, or the days Monday through Friday. To specify a subrange type, you must identify the smallest and the largest value in the subrange. A subrange *type-spec* has the syntax:

ordinal-constant . . ordinal-constant

where the two *ordinal-constants* are constants of the host type. The first constant specifies the lower bound of the subrange, and the second, the upper bound. The subrange is a closed interval—that is, the bounds are included. The first constant must not be greater than the second constant.

A subrange of integer values must lie exclusively within either the INTEGER, WORD, or LONGINT range; the subrange may not be a combination of the three ranges.

A variable of a subrange type possesses all the properties of a variable of the host type, with the restriction that its value must lie within the specified range.

Examples of valid subrange *type-specs*:

```
1..100
-10..+10
Monday..Friday
0..65535
```

The third example assumes that an enumerated type including the constants **Monday** and **Friday** has already been defined—for instance, the third example in the preceding section.

Examples of invalid subrange *type-specs*:

```
'z'..100
99..0
```

In the first example, the subrange bounds are of incompatible types. The second example has the lower bound greater than the higher bound.

Subrange types allow you to specify bounds on the values in your program when the problem so dictates. The compiler can then perform range checking to help detect errors. If the bounds are exceeded during a program run, an error will be reported if you compiled your program using the CHECK control (10.4.1). Note that you cannot specify bounds for real numbers, since real types are not ordinal.

Real Types

A *real type* is a simple type that represents a real (floating-point) number. There are many real numbers (limited only by the discreteness of internal computer representations) whose values fall between two given real numbers. Hence, it is impossible to assign meaningful integral sequence numbers to the real numbers. Certain operations that can be performed on variables of ordinal types are invalid when applied to variables of a real type. These include the ordinal functions ORD, LORD, WRD, CHR, PRED, and SUCC, which are described in 8.1.

Pascal-86 predefines three real types: REAL, LONGREAL, and TEMPREAL. The *type-spec* for these types are the identifiers REAL, LONGREAL, and TEMPREAL, respectively. You operate on values of a real type by using operators such as addition and comparison, and by invoking predefined procedures and functions.

REAL is a real type whose values are a subset of the real numbers. In Pascal-86, REAL values are single-precision floating-point numbers, which are always four bytes long and have 24 bits of precision. (Appendix H gives the range of REAL values.) Values of type REAL are denoted by real numbers (3.3.3) whose values fall within the defined subset.

LONGREAL is a real type whose values are a subset of the real numbers. In Pascal-86, LONGREAL values are double-precision floating-point numbers which are always eight bytes long and have 53 bits of precision. (Appendix H gives the range of LONGREAL values.) Values of type LONGREAL are denoted by real numbers (3.3.3) whose values fall within the defined subset.

TEMPREAL is a real type whose values are a subset of the real numbers. In Pascal-86, TEMPREAL values are extended-precision floating-point numbers, which are always ten bytes long and have 64 bits of precision. (Appendix H gives the range of TEMPREAL values.) Values of type TEMPREAL are denoted by real numbers (3.3.3) whose values fall within the defined subset.

5.3.2 Structured Types

Variables of a *structured type* are collections of values. A structured type is characterized by the types of its components and by its structuring method. Subject to certain rules and restrictions, component types of a structured type may themselves be structured, resulting in a structured type with several levels of structuring. For instance, you might declare an array of records, each of which contains another array.

There are four structuring methods, or *type constructors*, in Pascal: ARRAY, RECORD, SET, and FILE. A structured *type-spec* has one of the following forms:

[PACKED] *array-type*
[PACKED] *record-type*
[PACKED] *set-type*

```
[PACKED] file-type
identifier
```

where

array-type, *record-type,* *set-type,* *file-type,* are as defined in the following sections.

identifier is the name of a previously defined structured type.

The PACKED Prefix

By prefixing a structured type specification with the keyword PACKED, you direct the compiler to economize storage for variables of the specified type. Such packing will be done even at the price of some loss in efficiency of access, and even if this may increase the size of the compiled code needed to access components of the structure. Thus, your decision whether to use packed or unpacked structures will depend on which is more important to you: saving data space, or saving code space and increasing speed of access to structure elements. The internal representation of packed structures may differ from compiler to compiler.

If a type has several levels of structuring, an occurrence of the PACKED prefix affects only the level of the structured type whose definition it precedes. If a component is itself structured, the component's representation is packed only if the PACKED prefix occurs in the definition of the component type as well.

The predefined procedures PACK and UNPACK convert the format of unpacked array variables to packed, and vice versa. These procedures are described in 8.6.

Array Types

An *array type* is a structured type consisting of a fixed number of items, or components, that are all of the same type (called the *component type*). For instance, you might define an array of 100 integers. A component of an array is designated by an array selector, or index, which is a value belonging to the *index type*. The index type, which must be an ordinal type, is usually a programmer-defined scalar type or a subrange of the type INTEGER. Given a value of the index type, an array selector yields a value of the component type. The time needed for a selection does not depend on the value of the index; thus an array is a random-access structure.

The *array-type* specification defines the component type and the index type. Its syntax is:

```
ARRAY [ index-type [, index-type]... ] OF component-type
```

where

index-type is the type specification of an ordinal type.

component-type is any type specification.

Examples of one-dimensional *array-type* specifications:

```
ARRAY [1..100] OF REAL
ARRAY [Boolean] OF color
ARRAY [Monday..Friday] OF AppointmentSchedule
```

If the component type of an array type is also an array type, resulting in an array of two or more dimensions, you may use an abbreviated form of definition. In this form, the index type of the component and the index type of the array are enclosed within the same set of square brackets. For example:

```
ARRAY [Boolean] OF ARRAY [1..10]
      OF ARRAY [size] OF LONGINT
```

is equivalent to:

```
ARRAY [Boolean, 1..10, size] OF LONGINT
```

and:

```
PACKED ARRAY [1..10] OF PACKED ARRAY [1..8] OF Boolean
```

is equivalent to:

```
PACKED ARRAY [1..10, 1..8] OF Boolean
```

The term *string type* is a generic term for any type defined to be:

```
PACKED ARRAY [1..n] OF CHAR
```

where

n is an integer constant between 1 and MAXLONGINT.

In Pascal-86, a single-character constant may be used as a string constant of type PACKED ARRAY [1..1] OF CHAR, except in the RESET and REWRITE procedures (see 8.7.1 and 8.7.2).

In program statements, you refer to a component of an array variable by giving the variable's denotation (the name of the array variable, if it is not itself a component) followed by an index expression enclosed in brackets. This index expression must be assignment-compatible with the index type of the array type, as defined in 5.3.4. Section 5.4.2 gives syntactic details and examples.

Record Types

A *record type* is a structured type consisting of a fixed number of components, possibly of different types. For instance, a record might consist of a person's name (a character string), height (an integer), and weight (an integer). Records in Pascal are similar to structures in ASM86 and in PL/M.

Components of a record, called *fields*, are selected by means of unique identifiers, called *field identifiers*, which are defined in the record type specification. As with arrays, the time needed to access a selected component does not depend on the selector, so a record is a random-access structure.

A record type may be specified as consisting of several *variants*. The presence of variants implies that different variables, although of the same record type, may assume structures that differ in a certain manner. The difference may consist of a different number and different types of components. For example, one variant of a record type might consist of a person's name, height, weight, and year of birth; another variant might consist of the person's name, height, weight, sex, and place of birth.

The variant which is assumed by the current value of a record variable may be indicated by a component field which is common to all variants and is called the *tag* *field*. Usually, the part common to all variants will consist of several components, including the tag field.

The syntax of a *record-type* specification is:

```
RECORD
[field-list [;] ]
END
```

where *field-list* is either:

```
field-id [, field-id] . . .  :  type-spec [;
field-id [, field-id] . . .  :  type-spec] . . .
```

or:

```
[field-id [, field-id] . . .  :  type-spec [;
field-id [, field-id] . . .  :  type-spec] . . .   ;]
CASE [field-id :] tagtype-id OF
case-const [, case-const] . . .  :  ( [field-list [;] ] ) [;
case-const [, case-const] . . .  :  ( [field-list [;] ] )] . . .
```

In this syntax, each *field-id* (field identifier) is a unique identifier, distinct from all other *field-id*'s at the same level in the record specification. The *type-spec* for each record component can be any *type-spec* defined in the program, including another record type specification. The portion of the syntax between the keywords CASE and OF is called a *tag*; the *tagtype-id* is the identifier of any ordinal type except LONGINT. Each *case-const* is a unique ordinal constant of a type that is compatible with the tag type. The set of case constant values need not equal the set of values in the tag type.

The occurrence of an identifier as a *field-id* is a definition of the identifier as a field identifier, which is the name of a component of a record. The scope of this definition is as follows:

- All field designators that contain a record variable whose type is the record type in which the *field-id* occurs, plus
- The *statement* of each WITH statement that specifies a record variable whose type is the record type in which the *field-id* occurs

To refer to a component of a record variable in program statements, you use a *field designator*. A field designator consists of the record variable's denotation (the name of the record variable, if it is not itself a component), followed by the field identifier of the component. Section 5.4.2 gives syntactic details and examples.

The WITH statement, described in 7.2.9, allows you to use a shorter notation in referring to components of a record variable: in the *statement* embedded inside the WITH statement, you need only name the individual field identifiers rather than the full field designators.

Examples of *record-type* specifications for simple (non-variant) record types:

```
RECORD day:   1..31;
       month:  1..12;
       year:  integer;
END
```

```
RECORD node:  PACKED ARRAY [1..20] OF CHAR;
       leftbranch, rightbranch:  ↑TreeElement;
END
```

In the first example, the field identifiers are **day**, **month**, and **year**; in the second, they are **node**, **leftbranch**, and **rightbranch**. If you declare a record variable of the first type and call it TODAYSDATE, for example, you refer to the fields of that variable using the field designators TODAYSDATE.DAY, TODAYSDATE.MONTH, and TODAYSDATE.YEAR.

The second form of the *field-list* in the syntax denotes a variant record type. The variant part begins with the keyword CASE. Note that this form of the syntax is defined recursively, since it contains a *field-list*. The variant part provides for the specification of a tag type with an optional tag field (*field-id*). If present, the tag field contains a value indicating which variant is assumed by the record variable at a given time. An error occurs if you make a reference to a field of a variant other than the current variant; however, the Pascal-86 compiler does not detect this error.

Each variant is introduced by one or more constants. All the case constants are distinct, and are of an ordinal type that is compatible (5.3.4) with the tag type. The set of case constant values need not equal the set of values in the tag type.

For a record with a tag field, a change of variant occurs only when your program assigns to the tag field a value associated with a different variant. At that moment, fields associated with the previous variant cease to exist, and those associated with the new variant come into existence with undefined values.

For a variant record without a tag field, a change of variant occurs when your program performs an assignment to a field that is associated with a new variant. Again, fields associated with the previous variant cease to exist, and those associated with the new variant come into existence with undefined values.

Examples of *record-type* specifications with variants:

```
RECORD x, y: real;
       area: real;
       CASE s: shape OF
          triangle:   (side: real;
                       inclination, angle1, angle2:
                       angle);
          rectangle:  (side1, side2: real;
                       skew, angle3: angle);
          circle:     (diameter: real);
END

RECORD title: alfa;
       CASE p: pubtype OF
          book:    (author, publisher: alfa;
                    copyrightdate: integer);
          album:   (label: alfa;
                    recordingdate: integer;
                    CASE recordingtype OF
                     popular:       artist: alfa;
                     classical:    (orchestra: alfa;
                                    conductor: alfa);
                     spoken:       (narrator: alfa;
                                    humorous: Boolean));
END
```

Set Types

A *set type* is a structured type consisting of a collection of objects. Examples are the set of alphanumeric characters, the set of positive integers less than or equal to 100, and the set of ingredients (out of a finite number of possible ingredients) called for in a recipe. Pascal defines a number of operators—such as union, intersection, and inclusion—on set-type operands. These are described in 7.1.6 and 7.1.7.

In Pascal, all members of a set must be of the same type, called the *base type*, which must be any ordinal type except WORD or LONGINT.

A set type specification, or *set-type* in the syntax, defines all the values that are possible members of a set of that type. In mathematical terms, it defines the power set, or collection of all subsets, of the base type (recall that a type is itself a set, that is, a collection of permissible values). Its syntax is:

```
SET OF ordinal-type
```

where

ordinal-type is the base type. It may be the name of a predefined or userdefined simple, enumerated or subrange type (except a subrange of type WORD or LONGINT), or the type specification of an enumerated or subrange type. Restrictions on set elements in Pascal-86 are given in Appendix C.

Examples of *set-type* specifications are:

```
SET OF 1..1000
SET OF CHAR
SET OF color (* color = (red, yellow, blue, green,
  orange, violet) *)
```

The value of an object of a set type is denoted by listing all its members within brackets, separated by commas. The elements listed in brackets, which may themselves be expressions, all must be of the base type. (These values may be of type WORD or LONGINT only if they lie in the range −32767 to +32767.) The notation [] denotes the *empty set*, which contains no elements and belongs to every set type. The set [X..Y] denotes the set of all values of the base type within the closed interval from X to Y. For instance, [1..100] denotes the set of all integers from 1 to 100. (Note the analogy, in both syntax and meaning, to a subrange type.) If X is greater than Y (in ordinal value), then [X..Y] denotes the empty set.

For example, the following are all permissible values for a variable of the set type **SET OF color** defined above:

```
[red .. blue]            [red, blue]    [blue]
[yellow, blue, red]      [blue, red]    []
[red..blue, orange]
```

Since a set is a collection of values in which order does not matter, the first two sets in the first column are equivalent; likewise, the two sets in the second column are equivalent.

If W and L are variables of type WORD and LONGINT whose constant values lie in the range 1 to 1000, then the following are all permissible values for a variable of the set type **SET OF 1..1000:**

```
[1..10]     [W..L]
[W]         [1..L, W]
```

The Pascal-86 predefined type AT87ERRORS, discussed in 8.10, is a set type.

File Types

File types in Pascal allow variables to correspond to physical files, such as disk files, in the world outside the program. Thus they are the means by which a Pascal program obtains input and output data.

You specify the files your program uses as parameters in the program heading at the beginning of your main program block. Within your Pascal program, your file type definitions and variable declarations are independent of the nature of the physical files with which they are associated. You specify the logical file/physical file association by using the file preconnection feature described in 12.4, or by naming the physical file explicitly as a second parameter to the file handling procedures REWRITE and RESET, as detailed in 8.7.

A Pascal program views a physical file as a variable of a *file type*, which is structured like a magnetic tape. A file type consists of a sequence of components, like blocks on a tape, that are all of the same type. The number of components, called the *length* of the file, is not fixed, so your program can add or delete components as it runs. You do this by appending components, one by one, to the end of the file. Only one component of a file is accessible at any time, and you can change the currently accessible component only by moving sequentially through the file. Thus a file is a sequential-access structure.

A file with zero components is called *empty*.

The syntax of a *file-type* specification is:

```
FILE OF type-spec
```

The *type-spec* can be any *type-spec* defined in this chapter, except a file type or a type that contains a file.

Examples of *file-type* specifications (given the sample type definitions in 5.3):

```
FILE OF CHAR
FILE OF person
FILE OF 1..10
FILE OF color
```

A file variable can be a text file or a non-text file. *Text files* are of the predefined file type TEXT, which has components of type CHAR and is substructured into lines terminated by a special sequence of characters called the *line marker*. In the Pascal-86 logical record system (K.3), the line marker is the ASCII two-character sequence carriage return and line feed. Pascal-86 predefines the identifiers CR and LF to stand, respectively, for these two characters. The predefined program parameters INPUT and OUTPUT are text files. *Non-text files* are any files not declared to be of type TEXT.

You can open a file for reading or writing, read or write one file component (one character for a text file), read or write a sequence of file components, and check for the end of an input file. On text files, you may also read a sequence of characters from a line and skip to the next line, write a line, check for the end of the current line, and write a form feed indicator to start a new page of printed output. You perform all these operations by using the predefined file procedures described in 8.7. Note that the nature of your physical file may impose restrictions; for instance, you cannot read from a file connected to a line printer or write to a file connected to a console keyboard.

In standard Pascal there is no procedure for closing a file. Files are closed automatically when execution returns from the program block. The procedure for closing a file in Pascal-86 is outlined in Appendix B.

Whenever you declare a file variable F with components of type T, this also causes the implicit declaration of a variable of type T, denoted by F ↑ or F@. The predefined file input and output procedures fill, empty, and test this variable, called the *buffer variable* of the file. As long as the value of the buffer variable is defined, you may use it in expressions just as you would any other variable of type T. The syntax of the buffer variable is, as already suggested:

file-variable ↑ or *file-variable* @

where

file-variable is the name of a file variable. (The ↑ and @ are interchangeable.)

Sections 8.7, 8.2.2, and 8.2.3 give a fuller explanation of file input and output operations, and the sample programs in Chapter 9 illustrate these operations.

5.3.3 Pointer Types

A *pointer type* is a special type that represents an address—i.e., points to an area of storage. You use pointer types to allocate, access, and deallocate dynamic variables.

You generate dynamic variables within your program as you need them, rather than declaring them explicitly within a block. Dynamic variables are useful in creating complex data structures, such as linked lists and trees, that must change in size and form as the program runs.

The *type-spec* for a pointer type has the syntax:

↑ *type-id* or @ *type-id*

where

type-id is the identifier of a type (this may be another pointer type).

A pointer type thus consists of an unbounded set of values pointing to variables of the same type—the type specified in the syntax just given. (The ↑ and @ are interchangeable.)

You create a dynamic variable by calling the predefined procedure NEW with a pointer variable as an argument. NEW allocates a variable of that pointer's base type, and also assigns a value to the pointer so that it points to the newly allocated dynamic variable. The storage for dynamic variables is taken from a special pool of memory called the *heap*, which is provided through the Pascal run-time system. You cannot use the heap except as described here, by means of NEW, referenced variables, and DISPOSE.

Once you have allocated a dynamic variable, you access it as a *referenced variable* by referring to the associated pointer variable with the syntax:

pointer-id ↑ or *pointer-id* @

which is referred to as *dereferencing* the pointer, and is interpreted to mean "the variable pointed to by *pointer-id*," where *pointer-id* is the name of a pointer variable. (The ↑ and @ are interchangeable.) For example, if P is declared as a variable of type ↑T, then P denotes that pointer variable and its value, whereas P↑ denotes the dynamic variable of type T that is referenced by pointer P.

When you have finished using a dynamic variable, you may de-allocate it by calling the procedure DISPOSE. This causes the variable to become undefined and frees its storage on the heap for other dynamic variables.

Because of the special purpose of pointer types, no operations are defined on objects of this type except assignment and test for equality; that is, you can assign the value of one pointer variable to another or compare two pointer values, as long as they are both pointers to the same type.

Two pointer variables have the same type only if they are declared as follows:

```
VAR I,J: ↑INTEGER;
```

or:

```
TYPE INTEGER_PTR = ↑INTEGER;
VAR I: INTEGER_PTR;
    J: INTEGER_PTR;
```

Because each type specification defines a new and different type (5.3.4), when I and J are defined as:

```
VAR I: ↑INTEGER;
    J: ↑INTEGER;
```

they have different, and incompatible, types.

If two pointers are equal, the dynamic variables that they point to always occupy the same storage space. However, the internal representation of a pointer value is not defined by the language, so you should not attempt to work with it directly in your program.

The pointer value NIL belongs to every pointer type; it points to no object at all. You may assign the value NIL to a pointer to indicate, for instance, the end of a linked list.

Because pointer variables may also occur as components of structured variables which are themselves dynamically generated, the use of pointers permits you to represent any finite graph.

It is best to take special care in the use of dynamic variables in your programs. Programs using dynamic variables are more prone to logic errors than those using only static variables, and are generally more difficult to debug.

Sample Program 8 in Chapter 9 illustrates the use of dynamic variables.

5.3.4 Type Compatibility

This section gives the rules you must follow to make the types of your program objects consistent. It defines two terms used in many other sections of this manual—compatible and assignment-compatible. As a foundation for these definitions, it first defines real, integer, and string types.

Two types may be said to be compatible, whereas an expression may be said to be assignment-compatible with a type. Assignment compatibility is the more important concept to remember when you are writing programs. The compatibility of types is the basis for defining assignment compatibility.

Section 6.4.7 defines the compatibility of parameter lists.

With one exception, each instance of a type specification in a Pascal program defines a unique type. The one exception is the type specification that consists of a single identifier, which serves only to refer to a type object that has already been defined and does not generate a new one. For example, given:

```
TYPE color = (red, yellow, blue, green,
                orange, violet);
colour = color;
```

the identifiers **color** and **colour** are synonymous names for the same type.

With few exceptions, you can combine data objects in computations only if they have the same type. This section defines terms used in later sections of this manual to describe those objects of different types that can be combined in given types of computations.

A type is an *integer type* if it is the same as any one of the predefined types INTEGER, WORD, or LONGINT, or is a valid subrange of these types.

A type is a *real type* if it is the same as one of the predefined types REAL, LONGREAL, or TEMPREAL. (TEMPREAL is also an internal format used to evaluate REAL and LONGREAL expressions, as defined in 7.1.)

A type is a *string type* if it is defined to be:

```
PACKED ARRAY [1..n] OF CHAR
```

where

n is an integer constant between 1 and MAXLONGINT.

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

- They are the same type.
- One is a subrange of the other, or both are subranges of the same type.
- They are set types having compatible base types, and are either both packed or both unpacked.
- They are string types with the same number of components.

For example, given the following type definitions:

```
TYPE inttype1  = INTEGER;
     inttype2  = 1..10;
     inttype3  = 11..20;

     colortype1 = (red, yellow, blue, green, orange,
                   violet, brown, black);
     colortype2 = (red..blue);
     colortype3 = (brown..black);

     settype1  = SET OF colortype1;
     settype2  = SET OF colortype2;
     settype3  = SET OF colortype3;

     card = ARRAY [1..80] OF CHAR;
     line = ARRAY [1..80] OF CHAR;
     alfa = PACKED ARRAY [1..80] OF CHAR;
     beta = PACKED ARRAY [1..80] OF CHAR;

     person = RECORD name, firstname: alfa;
                     age: integer;
                     married: Boolean;
                     father, child, sibling: ↑person;
                     END;
```

then **inttype1**, **inttype2**, and **inttype3** are compatible; **colortype1**, **colortype2**, and **colortype3** are compatible; and **settype1**, **settype2**, and **settype3** are compatible. The identifiers **card** and **line** denote different and incompatible types. On the other hand, **alfa** and **beta** denote different but compatible types, because they are string types. Note also that the types of the **father**, **child**, and **sibling** fields of a **person** record are not compatible with **person** pointers outside of **person** records, because the pointer type specification has not been named.

An expression E of type T2 is *assignment-compatible* with type T1 if any of the following statements are true:

- T1 and T2 are the same type, which is neither a file type nor a structured type that contains a file type in any of its substructures.
- T1 is a real type and T2 is an integer or real type, and the value of expression E is within the range specified by T1.
- T1 and T2 are compatible ordinal types, and the value of expression E is within the range specified by the type T1.
- T1 and T2 are compatible set types, and all the members of the set given by set expression E are within the range specified by the base type of T1.
- T1 and T2 are compatible string types.
- T1 is of type PACKED ARRAY [1...1] of CHAR, and T2 is a CHAR TYPE.

At any place where the rule of assignment compatibility applies, any one of the following situations causes an error:

- T1 and T2 are compatible ordinal types, and the value of expression E is not within the range specified by T1.
- T1 and T2 are compatible set types, and any member of set expression E is not within the range specified by the base type of set type T1.
- T1 is a real type and T2 is any real or integer type, and the value of expression E is outside the range specified by T2.

For example, given the following type definitions:

```
TYPE inttype1 = INTEGER;

     realtype1 = REAL;

     colortype1 = (red, yellow, blue, green, orange,
                   violet, brown, black);
     colortype2 = (red..blue);
     colortype3 = (brown..black);
     colourtype1 = colortype1;

     settype1 = SET OF colortype1;
     settype2 = SET OF colortype2;
     settype3 = SET OF colortype3;
```

Then expressions of type **colortype1** are assignment-compatible with type **colourtype1**, and vice versa. Expressions of **inttype1** are assignment-compatible with **realtype1**, but expressions of **realtype1** are *not* compatible with expressions of **inttype1**. Expressions of **colortype2** and **colortype3** are assignment-compatible with **colortype1** and **colourtype1**, but expressions of **colortype2** are never assignment-compatible with **colortype3**, nor are expressions of **colortype3** assignment-compatible with **colortype2** (the two types are compatible, but their expression values cannot fall within the same subrange). Also, expressions of **settype2** and **settype3** are always assignment-compatible with **settype1**.

5.4 Variables

5.4.1 Variable Declarations

Variables are items of data whose values can change, and which the program manipulates. You must declare each variable in your Pascal program by means of a *variable declaration*, which assigns it a name and a type. The variable declarations in a Pascal block or non-main module appear in a list, separated by semicolons, following the keyword VAR (2.5.2).

The syntax of a variable declaration (*variable-decl*) is:

identifier [, *identifier*] . . . : *type-spec*;

where each *identifier* is unique, and *type-spec* is as defined in 5.3. The variable declaration defines all the identifiers in the list to be distinct variables of the given type. The scope of a variable declaration is the block in which the declaration occurs.

Assuming your program includes the sample type definitions given in 5.3, the following are all legal variable declarations:

```
VAR x,y: REAL;
    u,v: complex;
    z: TEMPREAL;
    i,j: INTEGER;
    k: 0..9;
    l: LONGINT;
    p,q: BOOLEAN;
    operator: (plus, minus, times);
```

```
a: ARRAY [0..63] OF REAL;
b: ARRAY [color, BOOLEAN] OF complex;
c: color;
f: FILE OF CHAR;
hue1,hue2: SET OF color;
p1,p2: ↑person;
```

5.4.2 Variable Denotations

When you use variables in expressions and statements in your program, you may designate an entire variable, a component of a structured variable, or a referenced variable.

Entire Variables

An *entire variable* is denoted by its identifier. Thus its syntax is simply:

identifier

For instance, if you declared a REAL variable **x** as in 5.4.1, you would use the identifier **x** to stand for the variable in expressions and statements.

Components of Array Variables

A component of an array variable is denoted by an *indexed variable*, which is the array variable's denotation followed by an index expression enclosed in brackets. If the array variable is itself a component of an array variable, you may use an abbreviated form in which both index expressions are enclosed in the same set of brackets.

Thus the syntax of an indexed variable is:

array-variable [*expression* [, *expression*] . . .]

where *array-variable* is the denotation of an array variable, and each *expression* is an index expression of any ordinal type. The index expressions need not have the same type as the corresponding index type of the array, though the two must be assignment-compatible (5.3.4).

Assuming that your program makes the sample variable declarations given in 5.4.1, the following are all legal denotations of components of array variables:

```
a [12]
a [i+j]
a [m-70000]
b [red]
b [red] [true]
b [red,true]
```

Note that **a[i+j]** is permissible only if the value of **i+j** is within the range 0 through 63. Likewise, **a[m-70000]** is permissible only if the value **m-70000** is within the range 0 through 63. The denotation **b[red]** stands for a one-dimensional array that is itself a component of the two-dimensional array **b**. The last two denotations, which are equivalent, stand for a single scalar component.

Components of Record Variables

A component of a record variable is denoted by a *field designator*, which consists of the record variable's denotation followed by a dot, followed by the field identifier of the component. The syntax of a field designator, therefore, is:

record-variable **.** *field-id*

where

record-variable is the denotation of a record variable.

field-id is the field identifier of the component.

Assuming the sample type definitions given in 5.3 and the sample variable declarations in 5.4.1, the following are examples of field designators:

```
u.re
b[red,TRUE].im
p2↑.size
```

Inside a WITH statement, you may use a shorter form—the field identifier only—to denote a record component, as described in 7.2.9.

Components of File Variables

At any time, only one component of a file variable may be referenced; the position of the file determines which component. This component is called the *current file component*, and is denoted by the *buffer variable* of the file. The syntax of a buffer variable is:

file-variable ↑ or *file-variable* @

where

file-variable is the denotation of the file variable. (The ↑ and @ are interchangeable.)

Hence, every declaration of a file variable F with components of type T implies the additional declaration of a variable of type T, denoted by F↑ or F@. For instance, assuming the sample variable declarations in section 5.4.1, the buffer variable f↑ exists and is of type CHAR. You use the buffer variable to append components to the file during generation and to access the file during inspection. To change the position of a file, you use the file input and output procedures described in 8.7.

An error occurs if you alter the position of a file F while the buffer variable F↑ is either an argument to a variable parameter or an element of the record variable list of an active WITH statement. However, the Pascal-86 compiler does not detect this error.

Referenced Variables

A dynamic variable you allocate using the predefined procedure NEW (8.5.1) exists until you deallocate it with the predefined procedure DISPOSE (8.5.2). Once you have allocated a dynamic variable, you access it as a *referenced variable* by referring to the associated pointer variable, with the syntax:

pointer-variable ↑ or *file variable* @

which is interpreted to mean "the variable pointed to by *pointer-variable*," where *pointer-variable* is the denotation of a pointer variable. (The ↑ and @ are interchangeable.) For example, if P is declared a variable of type ↑T, then P denotes that pointer variable and its pointer value, whereas P↑ denotes the variable of type T that is referenced by pointer P.

An error occurs if the pointer value is NIL (pointing to no value) or undefined at the time you use it to reference a dynamic variable.

Assuming the sample type definitions given in 5.3 and the sample variable declarations in 5.4.1, the following examples of referenced variables are valid:

```
p1↑.father
p1↑.sibling@.child
```

CHAPTER 6
PROCEDURES AND FUNCTIONS

6.1 Basic Concepts

Procedures and functions are subprograms that are contained within your main program and that may be nested within one another. They are callable from your main program, from containing procedures and functions, and from themselves. You activate a procedure (that is, invoke it or transfer control to its statement part) by using a procedure statement. You activate a function by referencing it in an expression within any statement.

A procedure or function declaration consists of a heading and either a block or the directive FORWARD. The heading consists of an identifier, by which the procedure or function is referenced, and a list of parameters.

The block consists of a compound statement which may contain embedded statements. The block also may include a set of constant definitions, type definitions, variable declarations, and additional procedure and function declarations. The statement part of the block specifies the actions to be performed when the procedure or function is invoked.

As described in 4.1.2, the constants, types, variables, procedures, and functions defined or declared in the block can be referenced only within the procedure or function itself, and are therefore called *local* to the procedure. Their identifiers refer to them only within the program text that constitutes the procedure or function declaration, which is called the *scope* of the local declarations and definitions. Since you may declare procedures and functions local to other procedures and functions, scope may be nested. Entities that you declare at the outer level of a module, i.e., not local to any procedure or function, are called *global.*

A procedure or function has a fixed number of parameters, each denoted within the procedure by an identifier. These *parameters* denote program objects through which the procedure or function communicates with its environment. When you invoke the procedure or function, you must indicate an actual quantity, or *argument*, to correspond to each parameter.

You declare functions analogously to procedures; the only difference is that a function yields a result, which you must specify in the function declaration. You may therefore use functions as operands in expressions. In order to eliminate side effects, you should àvoid making assignments to non-local variables and variable parameters within functions.

6.2 Procedure Declarations

A *procedure declaration* associates an identifier with a part of a program so that you may activate it via procedure statements, as described in 7.2.2. Its syntax is:

```
PROCEDURE identifier[( parameter-list )] ;
block
```

or:

```
PROCEDURE identifier[( parameter-list )] ;
FORWARD
```

The *identifier* in the procedure heading, which must be unique, is the name you use in a procedure statement to invoke the procedure. The syntax of a *parameter-list* is given in 6.4.1, and the syntax of a *block* is specified in 2.1. For an explanation of the second form of the procedure declaration, using the FORWARD directive, see 6.5.

The scope of the declaration of a procedure identifier is the block in which the procedure declaration occurs.

When you use the procedure identifier in a procedure statement within the procedure's *block*, you cause the procedure to be executed recursively.

The following are examples of procedure declarations:

```
procedure Product(var ProdMatrix : Matrices;
                  OneMatrix,TwoMatrix : Matrices);
var   I, J, K, Result : integer;
begin
  for I := 1 to MatrixSize do
    for J := 1 to MatrixSize do
      begin
        Result := 0;
        for K := 1 to MatrixSize do
          Result := Result + OneMatrix[I,K] * TwoMatrix[K,J];
        ProdMatrix[I,J] := Result
      end
end;  (* Product *)

procedure BuildTree;
var  FindRoot : Boolean;

    procedure AddNode;  (* add a node to the tree *)
    begin
      write(NodeCharacter : 3, NodeIndex : 3);
      with ExpressionTree[NodeIndex] do
        begin
          Symbol := NodeCharacter;
          Read(DataFile,Left); write(Left : 3);
          read(DataFile,Right); write(Right : 3);
          writeln
        end
    end;  (* AddNode *)

begin
  FindRoot := false;
  writeln('INPUT IS:');    writeln;
  AddNode;
  repeat
    readln(DataFile);
    read(DataFile,NodeCharacter,NodeIndex);
    if NodeIndex = 1 then FindRoot := true
    else AddNode
  until FindRoot or eof(DataFile);
  writeln
end;  (* BuildTree *)
```

The first procedure, which multiplies two matrices, has three parameters, all of type **Matrices**. The type **Matrices**, an array of integers, must be defined in the enclosing block. (Section 9.6 gives the complete program.) The second procedure, **BuildTree**, builds a binary tree. This procedure contains an embedded procedure, **AddNode**.

Neither procedure has any parameters. (Section 9.2 gives the complete program, which was first introduced in Chapter 2.)

6.3 Function Declarations

A *function declaration* designates a part of a program that computes a value. You activate a function by referencing it by name in function designators in expressions in your program, as described in 7.1.3. Its syntax is:

FUNCTION *identifier*[(*parameter-list*)] : *type-id* ;
block

or:

FUNCTION *identifier*[(*parameter-list*)] : *type-id* ;
FORWARD

The *identifier* in the function heading, which must be unique, is the name used in an expression to invoke the function. The *type-id* specifies the type of the result returned by the function; this result type must be a scalar or pointer type. The syntax of a *parameter-list* is given in 6.4.1, and the syntax of a *block* is specified in 2.1. Section 6.5 gives an explanation of the second form of the function declaration, using the FORWARD directive.

The scope of the declaration of a function identifier is the block in which the function declaration occurs.

Within the statement part of the block in a function declaration, there must be at least one assignment statement that assigns a value to the function identifier. The result of the function is the last value so assigned. If no assignment occurs, the value of the function is undefined. The function identifier serves within the function block as a structured variable identifier, which may be used in denotations for the components of the result.

Inside the function's *block* you may reference the result variable only on the lefthand side of an assignment statement. All other occurrences of the function identifier within its function block cause the function to be executed recursively.

The following are examples of function declarations:

```
FUNCTION Sqrt(x: REAL): REAL;
CONST eps = 0.0001;
VAR x0, x1: REAL;
BEGIN (* Newton's Method *)
    x1 := x;
    REPEAT
        x0 := x1;
        x1 := (x0+x/x0)*0.5;
        UNTIL abs(x1-x0) < eps*x1;
    Sqrt := x0;
END
```

```
FUNCTION RtoI(x: LONGREAL; i: INTEGER) : LONGREAL; (* I >=0 *)
VAR z: LONGREAL;
BEGIN
   z := 1;
   WHILE i > 0 DO BEGIN
      IF ODD(i) THEN z := z*x;
      i := i DIV 2;
      x := SQR(x);
      END;
   RtoI := z; (*z = x**i *)
END
```

The first function computes the square root of a real number using Newton's Method. The single parameter and the returned value are of type REAL. The second function computes x^i.

6.4 Parameters

The *parameters* given in the parameter list in a procedure or function heading are objects providing communication between the procedure or function and its environment. There are four kinds of parameters: value parameters, variable parameters, procedural parameters, and functional parameters. Value parameters are evaluated once, at invocation time, and the procedure or function can use the value but cannot change the argument. The arguments for value parameters may be expressions. Arguments for variable parameters must be variables, and the procedure or function may change their values. For procedural and functional parameters, the argument is a procedure or function identifier.

6.4.1 Parameter List Syntax

The syntax of a *parameter-list* is:

parameters [; *parameters*] . . .

where *parameters* has one of the following forms:

identifier [, *identifier*] . . . : *type-id*

VAR *identifier* [, *identifier*] . . . : *type-id*

PROCEDURE *identifier* [, *identifier*] . . . [(*parameter-list*)]

FUNCTION *identifier* [, *identifier*] . . . [(*parameter-list*)] : *type-id*

Each *identifier* in the syntax of *parameters* must be unique. In the first two forms of *parameters*, *type-id* may denote any type. In the last form, *type-id* must denote a simple type.

In a parameter list, a *parameters* group written without an initial special symbol (the first form shown) specifies that the constituent *identifiers* denote value parameters. Variable, procedural, and functional parameters are specified by a prefix of VAR, PROCEDURE, or FUNCTION, respectively.

When an *identifier* appears in a *parameters* specification for a value or variable parameter, it is defined as a parameter identifier within the *parameter-list* immediately containing it, and also as a variable identifier for the corresponding procedure or function block, if any.

When an *identifier* appears in a *parameters* specification for a procedural parameter, it is defined as a parameter identifier within the *parameter-list* immediately containing it, and also as a procedure identifier for the corresponding procedure or function block, if any.

When an *identifier* appears in a *parameters* specification for a functional parameter, it is defined as a parameter identifier within the *parameter-list* immediately containing it, and also as a function identifier for the corresponding procedure or function block, if any.

NOTE

> If the *parameter-list* is within a procedural or functional parameter specification, there is no corresponding procedure or function block.

The following are examples of parameter lists:

```
VAR f: TEXT

x: REAL;   i: INTEGER

FUNCTION f(x: REAL): REAL;
    a,b: REAL;
    VAR z: REAL
```

6.4.2 Value Parameters

The argument corresponding to a *value parameter* is evaluated once, at the time the procedure or function is invoked; the procedure or function can use the value but cannot change the argument. The argument may be any expression of the proper type.

The parameter represents a local variable within the procedure or function. At invocation, when the argument is evaluated, this value is assigned to the local variable before the procedure or function is executed. Thus a value parameter serves as an input to the procedure or function, but not as an output.

The argument must be assignment-compatible with the type of the parameter, as described in 5.3.4; thus you may not use file-type value parameters. Arguments passed to a value parameter of type LONGINT are computed using LONGINT arithmetic (7.1.4).

6.4.3 Variable Parameters

The argument corresponding to a *variable parameter* may be changed within the procedure or function. The argument to a variable parameter must be a variable (which may be a component of a structured variable), and the parameter represents this variable during the execution of the block. Any operation involving the parameter is performed directly on the argument variable. Thus a variable parameter serves as both an input to, and an output from, the procedure or function.

The argument and the parameter must be of the same type as defined in 5.3.4. If the selection of the argument variable involves indexing an array or referencing a dynamic variable (dereferencing a pointer), these actions are performed before the activation of the block.

The argument to a variable parameter may not be a component of a packed structure or array.

6.4.4 BYTES Parameters

A variable parameter of the predefined type BYTES lets you call external routines written in languages that are not as strongly typed as Pascal. Essentially, the BYTES type directs the Pascal-86 compiler not to type-check these parameters.

The argument in a procedure or function invocation corresponding to a BYTES parameter may be a variable, function, or procedure of any type. It may also be a real or string constant.

The BYTES type is particularly useful if you want to interface to general-purpose operating system routines and utility packages.

For example, most operating systems have a READ routine that reads an arbitrary number of characters into a buffer of any type. Standard Pascal would require that this READ routine be defined to read an arbitrary number of characters into a buffer of a specific Pascal type. All calls to the READ routine would have to pass a buffer of this specific Pascal type. Consequently, the generic nature of the operating system has been lost in the translation to Pascal.

```
TYPE BUFFTYPE = ARRAY[1..1024] of CHAR;
PROCEDURE READ(LEN: INTEGER; VAR BUFFER: BUFFTYPE);
```

The BYTES type can be used to restore the generic nature of the READ routine by allowing a buffer of any type to be passed (including BUFFTYPE).

```
PROCEDURE READ(LEN: INTEGER; VAR BUFFER: BYTES);
```

The BYTES type identifier may appear only as a variable parameter type in procedure or function headings in an external module's PUBLIC section (see 4.2). BYTES may not be used in procedure or function definitions in the PUBLIC section of the module currently being compiled, or in any local procedures or functions. Thus, Pascal-86 procedures cannot manipulate a parameter or variable of type BYTES.

Passing a procedure or function as an argument to a BYTES parameter is similar to passing a procedure or function as an argument to a FUNCTION or PROCEDURE parameter. FUNCTION and PROCEDURE parameters include the context of the argument procedure or function, as well as a pointer to the code (see Appendix J). BYTES parameters only get a pointer to the code. Since the context information is needed for the proper execution of nested procedures, a warning is issued if a nested procedure is passed as an argument to a VAR BYTES parameter. Nested procedure may not have type BYTES.

NOTE

BYTES parameters are permitted to allow an escape window in the type structure of Pascal-86. They should be used with caution.

6.4.5 Procedural Parameters

A *procedural parameter* allows you to write a procedure or function that itself invokes a variety of different procedures.

The argument corresponding to a procedural parameter is a procedure identifier, and the parameter denotes the argument procedure during the entire activation of the block. If the argument procedure, upon its activation through the parameter identifier, accesses any non-local object, the object accessed is the one that was accessible to the procedure when its procedure identifier was passed as a procedure argument.

The predefined procedures described in Chapter 8 cannot be used as procedural parameters. Interrupt procedures, discussed in 10.4.9, also cannot be used as procedural parameters.

The argument and parameter procedures must have compatible parameter lists, as defined in 6.4.7.

Procedural parameters are analogous to functional parameters, which are described in the next section. An example of a routine using a functional parameter appears at the end of that section.

6.4.6 Functional Parameters

A *functional parameter* allows you to write a procedure or function that itself invokes a variety of different functions.

The argument corresponding to a functional parameter is a function identifier, and the parameter denotes that argument function during the entire activation of the block. If the argument function, upon its activation through the parameter identifier, accesses any non-local object, the object accessed is the one that was accessible to the function when its function identifier was passed as a function argument.

The predefined functions described in Chapter 8 cannot be used as functional parameters.

The argument function and the parameter function must have compatible parameter lists (6.4.7) and the same result type.

For example, the following procedure computes the integral of a given function between limits A and B, using the trapezoidal rule with eight intervals:

```
PROCEDURE integrate (FUNCTION f(X:REAL): REAL;
                     a,b,: REAL;
                     VAR integral: REAL);


CONST n = 8;
VAR w, sum: REAL;
    i: INTEGER;
BEGIN
```

```
   w := (b-a)/n;
   sum := (f(a)+f(b))/2;
   FOR i:=1 TO n-1 DO
      sum := sum + f(a+i*w);
   integral := sum*w
END
```

To invoke this procedure to integrate sin(x) (given a user-written sine function called "sine") between the limits -pi/2 and pi/2, you would use the procedure statement:

```
integrate(sine,-pi/2,pi/2,int)
```

In the resulting invocation of **integrate**, wherever **f** occurs **sine** will be substituted for **f**, and the **sine** function will be called.

6.4.7 Parameter List Compatibility

Two parameter lists are *compatible* if they contain the same number of parameters and if the parameters in corresponding positions match. Two parameters *match* if one of the following is true:

- They are both value parameters of the same type.
- They are both variable parameters of the same type.
- They are both variable parameters, and one of them is of the predefined type BYTES (6.4.4).
- They are both procedural parameters with compatible parameter lists.
- They are both functional parameters with compatible parameter lists and the same result type.

6.5 The FORWARD Directive

In standard Pascal, you must declare each procedure or function before you reference it in other parts of your program. When two or more procedures or functions are mutually recursive (e.g., A calls B and B calls A), declaration before reference is impossible to do with a single declaration for each procedure or function; so standard Pascal provides the forward declaration, which uses the FORWARD directive.

In the syntax for procedure and function declarations, you may use the directive FORWARD in place of the procedure or function block. FORWARD indicates that the block associated with the preceding heading appears later in the program text. Subsequently, when you introduce the associated block, you must precede it with a procedure or function heading having the same name as the forwarding declaration; but here, you may omit the parameter list and the return type (for a function). (In standard Pascal, you *must* omit them.) The forward declaration and the body declaration (the declaration containing the block) are local to the same containing block or module (their scope is that containing block or module), and together they constitute the declaration of the procedure or function identifier.

A procedure or function identifier is considered to be declared as soon as it is given in a procedure or function heading. Hence, a directly recursive procedure or function—one that calls itself within its statement part—does not need a forward declaration.

The following sample program fragment shows a forward declaration and its corresponding body declaration later in the program. Note that in this particular example the procedure shows only simple recursion, so the forward declaration is not really needed, but this simple example (examples of indirect recursion are more complex) illustrates the syntax.

```
FUNCTION GCD(m,n: INTEGER): INTEGER;

FORWARD;

FUNCTION max(A: arrayofreals): REAL;
   (* arrayofreals is defined in the enclosing block *)
VAR m: REAL; i: INTEGER;
BEGIN
   x := A[1];
   FOR i:=2 TO n DO IF x<A[i] THEN x:=A[i];
   max := x;
END;

FUNCTION GCD; (* no parameter list needed here *)
BEGIN
   IF n=0 THEN GCD:=m
      ELSE GCD:=GCD(n,m MOD n)
END;
```

To make your programs easier to follow, especially when doing top-down programming, you may find it useful to give FORWARD declarations for all your procedures and functions. Once you have done this, you can place the body declarations after the forward declarations in any order you wish.

FORWARD is not a reserved word, so you may use it as an identifier in your program. Unlike predefined identifiers, however, FORWARD does not lose its original meaning as a directive when you define or declare it as an identifier. In such a case, each time FORWARD appears, the context determines which of the two meanings (directive or identifier) applies.

NOTE

The Pascal-86 compiler does not check for non-standard forward references, so the FORWARD directive is not required in Pascal-86 programs. However, you should use it if you wish your programs to be portable.

CHAPTER 7 EXPRESSIONS AND STATEMENTS

Statements in Pascal describe the actions to be performed on the data in your programs. Statements may include *expressions*, which denote rules for generating new values by applying operators to operands. The expressions are evaluated during program execution as required by the statements that contain them.

7.1 Expressions

Expressions are combinations of *operands*—variables and constants of scalar or set types—and *operators* that use the operands to compute new values. An expression may include one or more matched pairs of parentheses which serve to group operands and operators as desired. You use function designators (references to functions) in the same way as operands; thus you may use functions, as well as operators, to operate on values. An error is caused if any variable or function you use as an operand in an expression has an undefined value at the time the expression is evaluated. You will probably use expressions most often in assignment statements (7.2.1), which assign a newly computed value to a variable or a component of a variable.

Pascal provides four kinds of operators:

- The *arithmetic operators*: addition, subtraction, negation, multiplication, division, and remainder. Results are of type INTEGER, WORD, LONGINT, or TEMPREAL.
- The *Boolean operators*: negation (NOT), disjunction (OR), and conjunction (AND). Results are of type BOOLEAN.
- The *set operators*: union, intersection, and set difference. Result types are sets based on ordinal types.
- The *relational operators*: equality, inequality, ordering, set membership, and set inclusion. Results are of type BOOLEAN.

No operators producing new arrays, records, or files are defined. You assign new values to these by changing their scalar components individually. In addition, you may transfer the values of entire structures among variables associated with the same explicit type definition.

The relative *precedence* of operators determines which ones are applied first when an expression is evaluated during program execution. Operators are of four levels of precedence. From highest to lowest, they are

1. The negation operator (NOT)
2. The *multiplying operators*: multiplication or set intersection (*), division (/), division with truncation (DIV), remainder (MOD), and conjunction (AND)
3. The *adding operators*: addition, unary identity, or set union (+); subtraction, unary negation, or set difference (−); and disjunction (OR)
4. The *relational operators*: equality (=), inequality (< >), less than (<), greater than (>), less than or equal to *or* set inclusion (≤), greater than or equal to *or* set inclusion (≥), and set membership (IN)

When an expression is evaluated upon execution of the statement that contains it, operators of highest precedence are applied first.

The order in which the operands of a binary operator are evaluated (if they are subexpressions or function designators, or involve indexing an array or dereferencing a pointer) is undefined, so you should make no assumptions about this order. If the order of evaluation of subexpressions is important, put them in separate statements.

7.1.1 Expression Syntax

An expression may be composed of subexpressions called simple expressions, terms, and factors. Syntactically, factors are combined via multiplying operators to form terms; terms are combined via adding operators to form simple expressions; simple expressions are combined via relational operators to form expressions.

By definition, simple expressions, terms, and factors are always expressions, too. Thus a variable identifier is an expression; so is a constant.

The syntax of an *expression* is:

simple-expression [relational-op simple-expression]

where

relational-op is a relational operator.

simple-expression is a simple expression with the syntax:

[sign] term [adding-op term]...

Here, *sign* is a plus or minus sign used as a unary operator, *adding-op* is an adding operator, and *term* is given by:

factor [multiplying-op factor]...

where

multiplying-op is a multiplying operator as defined in 7.1.

factor is any one of the following:

an entire variable, structured variable component, or referenced variable (5.4.2)
a named constant (5.2)
a literal integer (3.3.2)
a literal real number (3.3.3)
a literal string (3.3.5)
a function designator (7.1.3)
NIL (5.3.3)
(*expression*)
NOT *factor*
[*element* [, *element*]...] where *element* is given by *expression*[. . *expression*]

The last item in the list of forms for *factor* represents a set, and each *element* stands for one or more set members. If two expressions joined by ellipses are given, the *element* represents all the elements within the given subrange of the base type of the set.

The following are some examples of factors, terms, simple expressions, and expressions.

Factors:

```
x
15
(x+y+z)
abs(x+y)
[red,c,green]
[1,5,10..19,23]
NOT p
```

Terms:

```
x*y
i/(1-i)
p AND q
(x <= y) AND (y > z)
```

Simple expressions:

```
x+y
-x
hue1 + hue2
i*j+1
```

Expressions:

```
x = 1.5
p <= q
(i < j) = (j > k)
c IN hue1
```

Note that all the examples of factors, terms, and simple expressions are also expressions.

7.1.2 Operands

As defined in the syntax for an expression, an operand may be an entire variable, structured variable component, referenced variable, constant, literal value (integer, real number, or string), the pointer value NIL, or a sequence of set elements enclosed in brackets. It may also be a function designator, which itself directs an operation to be performed.

Note that some operations are defined only for certain operand types or forms. Also note that certain type conversions occur automatically when expressions are evaluated at run time.

Automatic Conversions from Subrange Type to Host Type

Any operand whose type is S, where S is a subrange of T, is treated as if it were of type T. Similarly, any operand whose type is SET OF S is treated as if it were of type SET OF T. Consequently, expressions of subrange types and set expressions based on a subrange type can never occur. Even an expression consisting of a single operand of type S is itself of type T, and an expression consisting of a single operand of type SET OF S is itself of type SET OF T.

Precision of Real Expressions

Any value whose type is REAL or LONGREAL is automatically converted to 80-bit extended precision, or TEMPREAL, format when it appears in an expression, and all arithmetic is performed in this extended precision. The result is converted to type REAL or LONGREAL, respectively, only upon assignment to such a variable.

For example, if **x**, **y**, and **z** are of type REAL, then the statement:

```
z := x*y
```

will be processed, in effect, by first converting both **x** and **y** to extended precision, then multiplying them and converting the extended precision result to type REAL upon assignment to **z**.

Section 7.1.8 discusses the TEMPREAL format in more detail.

Set Expressions

An expression (factor) consisting of a list of *elements* enclosed between square brackets represents a set, and is called a *set constructor*. The expressions that form the *elements* of a set constructor must be compatible with the base type of the set.

A left bracket immediately followed by a right bracket [ ] denotes the *empty set*, which contains no elements and belongs to every set type. The set [x..y] denotes the set of all values of the base type in the closed interval from x to y. If x is greater than y, then [x..y] denotes the empty set. An error occurs if the value of an expression that is a member of a set is outside the limits set by the compiler. In Pascal-86, the ordinal values of set members must be one of the integer types in the range −32767 through +32767.

For example, given the following type definitions:

```
TYPE color = (red, yellow, blue, green,
   orange, violet);
```

then the following are all permissible set expressions of type **SET of color**:

```
[red..blue]            [red, blue]  [blue]
[yellow, blue, red]    [blue, red]  []
[red..blue, orange]
```

Since a set is a collection of values in which order does not matter, the first two set expressions in the first column are equivalent; likewise, the two set expressions in the second column are equivalent.

If W and L are values of type WORD and LONGINT that lie in the range 1 to 1000, then the following are all permissible values for a variable of the set type **SET OF 1..1000:**

```
[1..10]     [W..L]
[W]         [1..L, W]
```

7.1.3 Function Designators

A *function designator* specifies the activation of the function denoted by the function identifier. You use a function designator in an expression as if it were an operand.

When the expression is evaluated, the function is activated and the result is computed and substituted in the expression.

A function designator may contain a list of arguments to be substituted in place of their corresponding parameters in the function declaration. The correspondence is established by the order of the items in the lists of arguments and parameters, respectively; the first argument matches the first parameter, and so on. The number of arguments must equal the number of parameters.

The order in which the arguments are evaluated and associated with their parameters may vary, so you should make no assumptions about this order.

The syntax of a function designator is:

functin-id [(*argument* [, *argument*]...)]

where *function-id* is a function identifier, and each *argument* is either an expression or a procedure or function identifier.

Examples:

```
Sum(a, 100)
GCD(147, k)
ABS(x-y)
EOF(f)
ORD(blue)
```

7.1.4 Arithmetic Operators

Table 7-1 summarizes the binary and unary arithmetic operators.

The symbols +, −, and * also denote operations on sets, which are defined in 7.1.6.

Subexpressions of INTEGER, WORD, and LONGINT types can be operands of the same expression. The minimum result type for each operation is determined according to table 7-2.

If one of the operands is of a real type (REAL, LONGREAL, or TEMPREAL), then both operands are converted to the TEMPREAL (extended precision) format, and arithmetic is performed in TEMPREAL precision. Results are converted to a real type upon assignment. Note that all real expressions are evaluated in TEMPREAL precision.

Arithmetic operations are performed on two integer values according to the following rules:

- If both operands are of the same predefined type, the result is of that type.
- If one operands is of type LONGINT, the result is of type LONGINT.
- If one operands is of type INTEGER and the other is of type WORD, the result is of type LONGINT.

If the unary negation operator is used on a WORD value, the result is of type LONGINT.

If the evaluation of any expression creates a result of type LONGINT, the LONGINT type will be propagated back through the expression until all operands are of type LONGINT. This helps to avoid run-time exceptions when evaluating the expression.

Table 7-1. Arithmetic Operators

Binary

Symbol	Operation	Type of Operands*	Type of Result*	Level of Precedence
+	addition	integer or real	integer or TEMPREAL	3
−	subtraction	integer or real	integer or TEMPREAL	3
*	multiplication	integer or real	integer or TEMPREAL	2
/	division	integer or real	TEMPREAL	2
DIV	division with truncation	integer	integer	2
MOD	remainder	integer	integer	2

Unary

Symbol	Operation	Type of Operand*	Type of Result*	Level of Precedence
+	identity	integer or real	integer or TEMPREAL	3
−	negation	integer or real	integer or TEMPREAL	3

*As used here, an integer type is one of the predefined types INTEGER, WORD, or LONGINT. A real type is one of the predefined types REAL, LONGREAL, or TEMPREAL. All real expressions are evaluated in TEMPREAL format and, if necessary, are converted to type REAL or LONGREAL upon assignment.

Table 7-2. Result Types of Mixed-Mode Arithmetic

	INTEGER	WORD	LONGINT	real
INTEGER	INTEGER	LONGINT	LONGINT	TEMPREAL
WORD	LONGINT	WORD	LONGINT	TEMPREAL
LONGINT	LONGINT	LONGINT	LONGINT	TEMPREAL
real	TEMPREAL	TEMPREAL	TEMPREAL	TEMPREAL

NOTE: Constants in the range 0 to 32767, and ordinal variables and functions with subrange types in the range 0 to 32767 are typed as either INTEGER, WORD, or LONGINT (matching the type of the other operand). If both operands lie in this range, they are typed as INTEGER.

Constants in the range 32767 to 65535 are typed as WORD only if the other operand is of type WORD. Otherwise, operands in this range are typed as LONGINT.

If your Pascal-86 program attempts an operation that would yield a value outside the range of the indicated result type, a run-time exception condition is caused. Section 14.2 discusses run-time arithmetic exceptions.

For nonzero values of j, i DIV j is equivalent to TRUNC(i/j) as defined in 8.4.1; for j=0, i DIV j causes an error. The operation i MOD j is illegal if j is zero or negative; otherwise the result is (i-(K*j)) for all K such that 0≤i MOD j <j. Note that the comparison i MOD j = i-(i DIV j)*j holds only if i ≥ 0.

Arithmetic on real types in Pascal-86 follows additional special rules, as described in 7.1.8.

7.1.5 Boolean Operators

Table 7-3 summarizes the Boolean operators in Pascal. These operate on BOOLEAN values and return BOOLEAN results.

The result of a *logical OR* is TRUE if one or both operands are TRUE, and FALSE otherwise. The result of a *logical AND* is TRUE if both operands are TRUE, and FALSE otherwise. The result of *logical negation* (NOT) is TRUE if the operand is FALSE, and FALSE if the operand is TRUE.

The values of some Boolean expressions (e.g., SWITCH1 OR ODD(N) when SWITCH1 = TRUE) can be determined by partial evaluation. The language does not define whether such expressions will be evaluated partially or completely.

7.1.6 Set Operators

Table 7-4 summarizes the set operators in Pascal. These operate on sets and return set results.

The *union*, or sum, of two sets A and B is the set of all items that are members of A, members of B, or both. The *intersection*, or product, of two sets A and B is the set of all items that are both members of A and members of B. The *difference* between set

Table 7-3. Boolean Operators

Binary				
Symbol	**Operation**	**Type of Operands**	**Type of Result**	**Level of Precedence**
OR	logical OR	BOOLEAN	BOOLEAN	3
AND	logical AND	BOOLEAN	BOOLEAN	2
Unary				
Symbol	**Operation**	**Type of Operand**	**Type of Result**	**Level of Precedence**
NOT	logical negation	BOOLEAN	BOOLEAN	1

Table 7-4. Set Operators

Binary				
Symbol	**Operation**	**Type of Operands**	**Type of Result**	**Level of Precedence**
+	set union	any set type T	T	3
−	set difference	any set type T	T	3
*	set intersection	any set type T	T	2

B and set A, also called the *relative complement* of A with respect to B and written B-A, is the set containing all items that are members of B but not members of A.

For example, if sets A and B have the following members:

```
A := [red, orange, yellow, green, blue, violet]
B := [red, blue, yellow, brown, black]
```

then the following relational expressions hold:

```
A+B = B+A
A*B = B*A
A+B = [red, orange, yellow, green, blue, violet,
            brown, black]
A*B = [red, yellow, blue]
A-B = [orange, green, violet]
B-A = [brown, black]
```

7.1.7 Relational Operators

Table 7-5 summarizes the relational operators in Pascal. These test relations between two operands and return BOOLEAN results.

The operands represented by the symbols =, <>, <, >, ≤, and ≥ are either of compatible types, or else one operand is of a real type and the other is of an integer type.

Table 7-5. Relational Operators

Binary				
Symbol	**Operation**	**Type of Operands**	**Type of Result**	**Level of Precedence**
=	equality / equivalence	any set, simple, string, or pointer type	BOOLEAN	4
<>	inequality / inequivalence	any set, simple, string, or pointer type	BOOLEAN	4
<	less than	any simple or string type	BOOLEAN	4
>	greater than	any simple or string type	BOOLEAN	4
≤	less than or equal to / logical implication / set inclusion (contained in)	any set, simple or string type	BOOLEAN	4
≥	greater than or equal to / logical implication / set inclusion (contains)	any set, simple or string type	BOOLEAN	4
IN	set membership	right operand: STE OF T; left operand: any type compatible with T	BOOLEAN	4

Except when applied to sets, the operators ≤ and ≥ mean less than or equal to and greater than or equal to, respectively. If P and Q are BOOLEAN operands, then P≤Q means P implies Q, and P≥Q means Q implies P (since FALSE < TRUE).

For instance, if A, B, and C are integers and P and Q are BOOLEAN variables such that A=1, B=3, C=9, P=FALSE, and Q=TRUE, then the following relational expressions hold:

```
(A+B)<C = TRUE
(P<=Q) = TRUE
(P=Q) = FALSE
((C<A)=Q) = FALSE
```

When you use the relational operators =, <>, <, >, ≤, and ≥ to compare operands of string type (i.e., PACKED ARRAY [1..n] OF CHAR), the results are determined by the lexicographical ordering of the character set; in Pascal-86, this character set is the ASCII collating sequence as given in Appendix G. Thus, if R, S, and T are strings of type PACKED ARRAY [1..10] OF CHAR and:

```
R := 'and       '
S := 'band      '
T := 'an        '
```

then:

```
((T<R) and (R<S)) = TRUE
S<>T = TRUE
S>T = TRUE
S>=T = TRUE
R<T = FALSE
```

If A and B are set operands, A=B means A is equivalent to B (i.e., has exactly the same members), and A<>B means A is not equivalent to B. A≤B means A is included in B, and A≥B means B is included in A.

The IN operator returns TRUE if the value of the operand is a member of the set; otherwise, it returns FALSE. If the right operand is a set of an integer type, the left operand may be an INTEGER, WORD, or LONGINT type whose value lies in the range −32767 through +32767. If the left operand is outside this range, an exception occurs.

For example, assume that A, B, and C are sets such that:

```
A := [red, orange, yellow, green, blue, violet]
B := [red, blue, yellow, brown, black]
C := [orange, green, violet]
```

then the following relational expressions hold:

```
(A=B) = FALSE
(A<>B) = TRUE
(C<=A) = TRUE
(A>=C) = TRUE
(violet IN A) = TRUE
(black IN A) = FALSE
(black IN (A+B)) = TRUE
```

7.1.8 Real Arithmetic

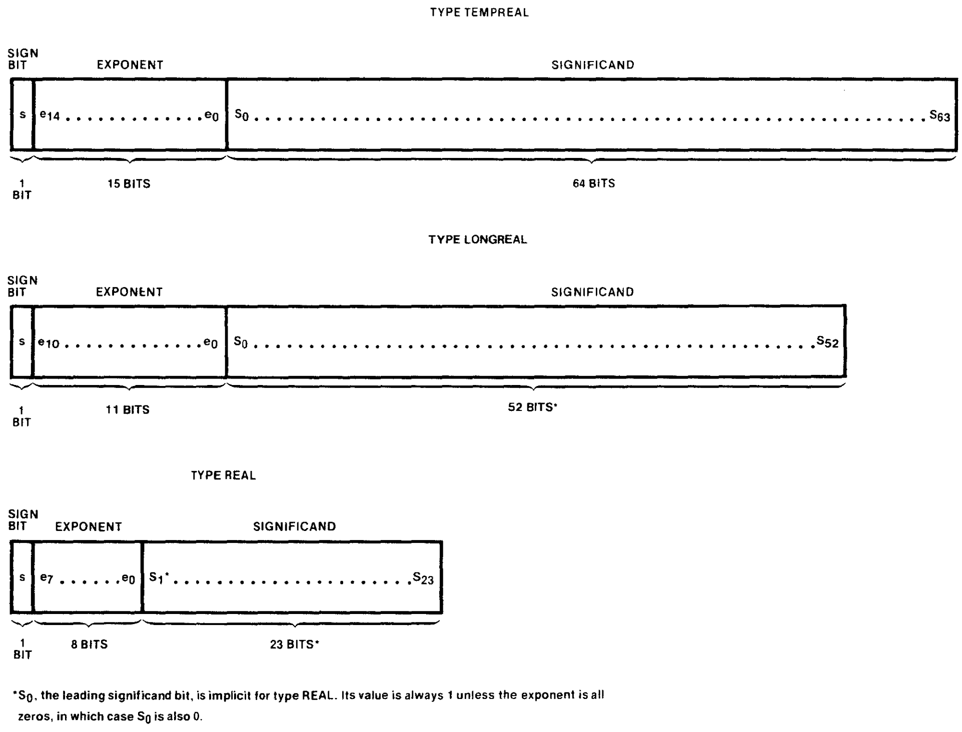
Real arithmetic in Pascal-86 conforms to the IEEE proposed standard for floating-point arithmetic. All run-time real arithmetic is performed on an 8087 Numeric Data Processor or by the 8087 software emulator provided as part of the Pascal-86 run-time system. Whether the 8087 processor or the 8087 emulator performs the computations is determined at link time, and depends upon which run-time libraries you link in (see Chapter 12). The results are independent of which one you select. Unless you wish to do specialized error handling, you need not be concerned with the 8087 processor or the 8087 emulator when writing Pascal programs.

This section presents the information you normally need to write Pascal-86 programs using real arithmetic. For further information you may need to determine the cause of real arithmetic errors, see 14.6 and 14.7. Complete information on the functionality and use of the 8087 processor and emulator is provided in the *iAPX 86,88 User's Manual.*

The discussion of real arithmetic is in this section and in 14.6 and 14.7 applies to the operations of real addition, subtraction, multiplication, and division, the predefined arithmetic and transfer functions (8.3, 8.4), and assignmments, input, and output involving real types.

Representation of Real Numbers

A real value v is represented in a binary floating-point format consisting of a *sign bit* s, a *biased exponent* e, and a *significand* S such that $v=(-1)^s \cdot S \cdot 2^e$ (see figure 7-1). The significand is always a non-negative value less than two. In Pascal-86, three


TYPE TEMPREAL
SIGN BIT
EXPONENT
SIGNIFICAND
s
e_{14} .... e_0
S_0 .... S_{63}
1 BIT
15 BITS
64 BITS
TYPE LONGREAL
SIGN BIT
EXPONENT
SIGNIFICAND
s
e_{10} .... e_0
S_0 .... S_{52}
1 BIT
11 BITS
52 BITS*
TYPE REAL
SIGN BIT
EXPONENT
SIGNIFICAND
s
e_7 ...... e_0
S_1* .... S_{23}
1 BIT
8 BITS
23 BITS*

*S_0, the leading significand bit, is implicit for type REAL. Its value is always 1 unless the exponent is all zeros, in which case S_0 is also 0.

Figure 7-1. Pascal-86 Real Data Types

121539-34

precisions are used for real numbers: the 24-bit precision used for REAL variables (23-bit significand with implicit leading bit, 8-bit exponent, plus sign bit, or 32 bits total), the 53-bit precision used for LONGREAL variables (52-bit significant with implicit leading bit, 11-bit exponent, plus sign bit, or 64 bits total), or the 80-bit precision used for TEMPREAL variables and for all intermediate results (64-bit significant, 15-bit exponent, plus sign bit, or 80 bits total). All expressions and subexpressions are computed in TEMPREAL precision, and values are converted to REAL and LONGREAL precision only when the final result is assigned to such a variable.

In addition to real values, the floating-point representation allows for values that are not actual numbers. These are called NaN's (for "Not a Number"). The presence of NaN's in a computation generally indicates a real arithmetic exception condition.

The greater range of the exponent in TEMPREAL format greatly reduces the likelihood of underflow and overflow, and eliminates roundoff as a source of error until the final assignment of the result is performed. These advantages arise because underflow, overflow, and roundoff errors are more probable for intermediate computations than for the final result. For example, an intermediate underflow result might later be multiplied by a very large factor, providing a final result of acceptable magnitude.

Real Arithmetic Errors

Six types of real arithmetic exceptions (conditions raised by run-time errors) are detected in Pascal-86: invalid operation, denormalized operand, zero divide, overflow, underflow, and precision. (All but the Denormalized Operand exception are specified in the IEEE floating-point standard.) For each exception, a status bit is set on any occurrence of the corresponding exception.

After the status bit is set, the action taken depends on the setting of the error mask in the 8087 chip or emulator. If the exception type is masked, the chip or emulator performs a generally appropriate fixup—such as loading zero or a NaN—and continues, leaving the status bit set for that exception. If the exception is unmasked, the chip or emulator generates an 8086 interrupt, which invokes the real arithmetic error handler. Pascal-86 provides a default error handler, which prints out an error message and terminates the program being run. When linked to EH87.LIB, this default handler may retry the operation and resume the task. If you wish, you may substitute your own error handler, following the instructions in Appendix K.

A Pascal-86 main module initializes the error mask to mask all exceptions except the Invalid Operation exception. This is the recommended setting, which will provide the least abrupt, most appropriate action for many Pascal-86 applications. (To implement the IEEE Standard normalizing mode, EH87.LIB must be linked in and the Denormalized Operand exception must be unmasked.)

The 8087 chip or emulator can recover from most exception conditions, but the Invalid Operation exception is usually fatal. Other exceptions from which the chip or emulator cannot recover will eventually result in an Invalid Operation exception.

NOTE

The 8087 emulator and the 8087 processor run-time interface libraries use interrupt level 16 for real arithmetic error handling. If you are using the emulator, you must reserve interrupt level 16 for that purpose. If you are using the 8087 processor, you must either (1) connect the 8087 processor to 8086 interrupt level 16, *or* (2) connect the 8087 processor to some other 8086 interrupt level, then link in a routine to redirect the interrupts.

If you do not wish to use the default error mask setting, you may set the error mask in some other way using the predefined procedure MASK8087ERRORS.

If you are using the default error mask setting, or any other setting in which one or more errors are masked, it is recommended that your program call the predefined procedure GET8087ERRORS (8.10.1) at the end of any significant computation sequence, to check for masked exceptions. If a masked exception occurred, you can find out where the error occurred by re-running the computation sequence with that exception unmasked by means of the predefined procedure MASK8087ERRORS (8.10.2). The Pascal-86 default exception handler will then print out a message and exit when the error occurs. When masked exceptions occur, the presence of NaN's, infinities, or unnormalized or denormalized numbers may provide clues to the nature of the problem. However, it is rare that such exceptions will occur, *and* cause results that cannot be relied upon, without leading to an Invalid Operation exception.

7.2 Statements

Statements denote algorithmic actions, and are said to be *executable*. A statement may be prefixed by a label (followed by a colon), which can be referenced by GOTO statements (7.2.10). The following is an example of a labeled statement:

```
100: C := SQRT(SQR(A)+SQR(B))
```

Statements in Pascal can be classified as simple statements and structured statements. A *simple statement* contains no other statements; a *structured statement* contains embedded statements. Simple statements may be assignment statements, procedure statements, GOTO statements, or empty statements. Structured statements may be compound statements, IF statements, CASE statements, WHILE statements, REPEAT statements, FOR statements, or WITH statements.

An *empty statement* consists of no symbols and performs no action; unless it is labeled, it does not result in any compiled code. Empty statements may be useful as placeholders during top-down development and debugging.

Also, the existence of the empty statement means that if you inadvertently insert a semicolon between two statements where one is not needed (i.e., after the last statement in a sequence of statements), your program will still compile and run correctly.

Statements may also be classified according to function, as listed below:

- Computing new values: assignment statement
- Procedure invocation: procedure statement
- Statement grouping: compound statement
- Conditional execution: IF and CASE statements
- Repetitive execution (looping): WHILE, REPEAT, and FOR statements
- Identification of record names in references to record types: WITH statement
- Unconditional branching: GOTO statement

The following sections describe the various Pascal statements in the order just given.

7.2.1 Assignment Statements

The *assignment statement* replaces the current value of a variable by a new value specified as an expression. Its syntax is:

variable := *expression*

where

variable is an entire variable, a component of a structured variable, or a referenced variable.

expression must be assignment-compatible (5.3.4) with the type of the *variable.*

If *variable* is of type LONGINT, then *expression* is computed as a LONGINT result (7.1.4).

Note that *variable* may be of any type, including an array or record. Thus you may use a single assignment statement to transfer all the values of an array or record variable to another array or record variable, provided the types of the two variables are assignment-compatible.

If the selection of the variable involves the indexing of an array or the dereferencing of a pointer, then whether these actions precede or follow the evaluation of the expression is undefined.

Examples:

```
x  :=  y+z
p  :=  (1<=i) AND (i<100)
i  :=  SQR(k) - (i * j)
hue1 := [blue,SUCC(c)]
l:=MAXINT*3
```

7.2.2 Procedure Statements

A *procedure statement* specifies execution of the procedure denoted by the procedure identifier. The procedure statement may contain a list of arguments to be substituted in place of the corresponding parameters in the procedure declaration. The correspondence is established by the order of the items in the lists of arguments and parameters, respectively; the first argument matches the first parameter, and so on. The number of arguments must equal the number of parameters.

The order in which the arguments are evaluated and associated with their parameters is undefined, so you should make no assumptions about this order.

The syntax of a procedure statement is:

identifier [(*argument* [, *argument*]...)]

where *identifier* is a procedure identifier, and each *argument* is either an expression or a procedure or function identifier.

The argument corresponding to a value parameter of type LONGINT will be computed as a LONGINT result (7.1.4).

Examples:

```
BuildTree
Product(a,n,m)
Bisect(fct,-1.0,+1.0,x)
```

7.2.3 Compound Statements

A *compound statement* specifies that its component statements are to be executed in the sequence in which they appear. Thus a compound statement groups several statements into a single statement. The keywords BEGIN and END act as statement brackets.

The syntax of a compound statement is:

BEGIN *statement* [; *statement*]... END

where

each *statement* is any statement described in this chapter.

You must always use a compound statement as the statement part of every block in your program. In other words, you must group the statements in each block into a single compound statement enclosed within BEGIN and END brackets.

Examples:

```
BEGIN z:=x; x:=y; y:=z END

BEGIN
   RESET(f);
   REWRITE(g);
   WHILE NOT EOF(f) DO
      BEGIN
         g :=f↑; PUT(g); GET(f)
      END
END
```

7.2.4 IF Statements

A *conditional statement* (IF or CASE statement) selects for execution one of its component statements. The *IF statement* includes a Boolean expression, which it evaluates to determine whether to execute the first, or the optional second, component statement.

The syntax of the IF statement is:

IF *expression* THEN *statement* [ELSE *statement*]

The *expression* following the IF must be of type BOOLEAN; each *statement* may be any statement.

NOTE

No semicolon is permitted between the first *statement* and the keyword ELSE.

If the Boolean expression has the value TRUE, the statement following the keyword THEN is executed. If the Boolean expression is FALSE, the action depends on the existence of the ELSE clause. If the ELSE clause is present, the statement following the keyword ELSE is executed; otherwise, an empty statement is executed (no action is performed).

The construct:

```
IF e1 THEN IF e2 THEN s1 ELSE s2
```

is equivalent to:

```
IF e1 THEN
      BEGIN
         IF e2 THEN s1 ELSE s2
      END
```

In other words, the ELSE belongs to the most recent IF. To associate an ELSE with the first IF, you can enclose the second IF in a compound statement:

```
IF e1 THEN
      BEGIN
         IF e2 THEN s1
      END
ELSE s2
```

Examples:

```
IF x<1.5 THEN z:=x+y ELSE z:=1.5
IF p1<>NIL THEN p1:=p1↑.father
```

7.2.5 CASE Statements

The *CASE statement* specifies the execution of one of a list of component statements, based on the value of an expression that serves as a selector. Each statement is prefixed by one or more constants, called *case constants*; the statement executed is the one whose case constant is equal to the current value of the selector.

If none of the case constants is equal to the value of the selector, then the action of the CASE statement depends on the presence of the OTHERWISE clause. If the OTHERWISE clause is present, the statement sequence following OTHERWISE is executed; if this clause is not present, an error is caused.

The syntax of the CASE statement is:

```
CASE expression OF
   [case-const [, case-const]... : statement ;]...
   case-const [, case-const]... : statement [;]

END
```

or:

```
CASE expression OF
   [case-const [ , case-const]... : statement ; ]...
   OTHERWISE statement [ ; statement]...
END
```

where *expression* must be of an ordinal type, each *case-const* must be distinct and of an ordinal type compatible with the expression, and each *statement* may be any Pascal statement. Restrictions on case constants in Pascal-86 are given in Appendix C.

In the following examples, **i** is a variable of an integer type, **x** is a REAL variable, and **operator** is a variable of an enumerated type that may have the value **plus**, **minus**, or **times**:

```
CASE operator OF              CASE i OF
  plus:    x := x+y;            1:   x : = TRUNC(x);
  minus:   x := x-y;            2:   x := ROUND(x);
  times:   x := x*y;            OTHERWISE writeln ('CASE ERROR');
END                           END
```

7.2.6 WHILE Statements

The *repetitive statements* (WHILE, REPEAT, and FOR) execute their component statements repeatedly. The WHILE statement executes its component statement repeatedly as long as a given condition remains satisfied. Each time through the loop, the condition is tested *before* the statement is executed. Thus the WHILE statement is most useful when the loop is not performed at all in some circumstances.

The syntax of the WHILE statement is:

```
WHILE expression DO statement
```

where

expression must be a BOOLEAN expression.

statement may be any statement. Since only a single *statement* is allowed, you may wish to use a compound statement here.

The *statement* is repeatedly executed while, prior to each execution, the value of the BOOLEAN expression is TRUE. If its value is FALSE at the beginning, the statement is never executed at all.

The statement:

```
WHILE b DO s
```

is equivalent to:

```
IF b THEN BEGIN
             s;
             WHILE b DO s
          END
```

Examples:

```
WHILE switch=TRUE DO
   BEGIN
      sample(input);
      IF ABS(input)>toler THEN fixup;
      IF operatorstop THEN switch=FALSE;
   END

WHILE NOT eof DO
   BEGIN
      read(number);
      sum := sum+number;
   END
```

7.2.7 REPEAT Statements

The *REPEAT statement* executes its component statements until the given condition is satisfied. Each time through the loop, the condition is tested *after* the statement is executed. This means that one iteration of the loop is always performed.

The syntax of the REPEAT statement is:

REPEAT *statement* [; *statement*]... UNTIL *expression*

where each *statement* may be any statement and *expression* must be a BOOLEAN expression.

The statement sequence between the symbols REPEAT and UNTIL is executed repeatedly until the Boolean expression is TRUE after some execution of the sequence.

The statement:

REPEAT *s* UNTIL *b*

is equivalent to:

```
BEGIN s;
   WHILE NOT b DO s;
END
```

Examples:

```
REPEAT k := i MOD j;
       i := j;
       j := k
   UNTIL j=0

REPEAT PROCESS(f↑); GET(f) UNTIL EOF(f)
```

7.2.8 FOR Statements

The *FOR statement* executes its component statement repeatedly while a given finite progression of values is assigned to a variable, which is called the *control variable* of the FOR statement. The FOR statement is useful when the number of iterations is known at the time of the first iteration.

The syntax of the FOR statement is:

```
FOR variable := expression TO expression DO statement
```

or:

```
FOR variable := expression DOWNTO expression DO statement
```

where *variable* is a local variable of an ordinal type, and the two *expression*s are assignment-compatible with this type. The *statement* may be any Pascal statement; since only one is permitted, you may wish to use a compound statement.

The control variable serves as a counter. The progression of values assigned to the control variable starts with the value of the first expression and ends with the value of the second expression. If the TO form of the statement is used, the values in the ordinal type of the control variable are stepped through in order; if the DOWNTO form is used, the values are stepped through in reverse order. On each iteration, the appropriate value is first assigned to the control variable, and then the statement is executed. If the starting value is greater than the ending value for the TO form, or if the starting value is less than the ending value for the DOWNTO form, the statement will never be executed.

An error is caused if the control variable is altered by the repeated statement or by any statement activated by the repeated statement. However, the compiler does not check for such alterations occurring in a procedure or function invoked from within the FOR loop. After a FOR statement is executed, the value of the control variable is defined only if the FOR statement is terminated by a GOTO statement leading out of it, in which case the control variable has the value it had at the time of the GOTO.

Apart from the restrictions just given, the FOR statement of the form:

```
FOR v := e1 TO e2 DO s
```

is equivalent to the statement sequence:

```
BEGIN
    temp1 := e1;
    temp2 := e2;
    IF temp1 <= temp2 THEN BEGIN
        v := temp1;
        s;
        WHILE v <> temp2 BEGIN
            v := SUCC(v);
            s;
            END
        END
END
```

Similarly, a FOR statement of the form:

```
FOR v := e1 DOWNTO e2 DO s
```

is equivalent to the statement sequence:

```
BEGIN
    temp1 := e1;
    temp2 := e2;
    IF temp1 >= temp2 THEN BEGIN
```

```
        v := temp1;
        s;
        WHILE v <> temp2 DO BEGIN
            v := PRED(v);
            s;
            END
        END
END
```

In both cases, **temp1** and **temp2** are auxiliary variables, of the host type of the variable **v**, that do not occur elsewhere in the program.

Examples of FOR statements:

```
FOR i:= 2 TO 63 DO
    IF a[i] > max THEN max:=a[i]

FOR i:=1 TO n DO
    FOR j:=1 TO n DO BEGIN
        x := 0;
        FOR k:=1 TO n DO
         x := x + m1[i,k] * m2[k,j];
        m[i,j] := x
        END

FOR c:=blue DOWNTO red DO display(c)
```

7.2.9 WITH Statements

The *WITH statement* allows you to access the components of a record variable as if they were simple variables. Inside the WITH statement, including its component statement, all the field identifiers of the given record variables are defined as variable identifiers. Thus the WITH statement effectively extends the scope of the field identifiers of the records listed, so that they may be accessed as simply and efficiently as local variables.

The syntax of the WITH statement is:

WITH *variable* [, *variable*]... DO *statement*

where each *variable* must be a variable of a record type, and *statement* may be any Pascal statement. Since only one statement is permitted, you may wish to use a compound statement here.

The statement:

```
WITH v1, v2, v3, ..., vn DO s
```

is equivalent to:

```
WITH v1 DO
    WITH v2 DO
        WITH v3 DO
            . . .
                WITH vn DO s
```

The record variables **v1** through **vn** may be embedded inside each other (representing records within records) or completely separate.

If the selection of a variable in the list following the keyword WITH involves the indexing of an array or the dereferencing of a pointer, then these actions are executed before the component statement is executed.

To illustrate the WITH statement, let us assume that the following type definitions and variable declarations are given:

```
TYPE date = RECORD day:    1..31;
                   month:   1..12;
                   year:   integer;
            END;
     name = RECORD lastname:
                      PACKED ARRAY [1..20] OF CHAR;
                   firstname:
                      PACKED ARRAY [1..20] OF CHAR;
                   middleinit:   CHAR;
            END;
     studentrec = RECORD studentname:   name;
                      birthdate:   date;
                  END;
VAR currentrec: studentrec;
    todaysdate: date;
```

Then the WITH statement:

```
WITH currentrec, studentname, birthdate DO
   BEGIN
      lastname := 'Smith               ';
      firstname := 'Susan               ';
      middleinit := 'K';
      day := 28;
      month :=5;
      year := 1948;
   END
```

is equivalent to:

```
currentrec.studentname.lastname
   := 'smith               ';
currentrec.studentname.firstname
   := 'Susan               ';
currentrec.studentname.middleinit := 'K';
currentrec.birthdate.day := 28;
currentrec.birthdate.month := 5;
currentrec.birthdate.year := 1948;
```

Likewise,the WITH statement:

```
WITH todaysdate DO
   IF month=12 THEN BEGIN
      month := 1; year := year+1;
      END
   ELSE month := month+1
```

is equivalent to:

```
IF todaysdate.month = 12 THEN BEGIN
   todaysdate.month := 1;
   todaysdate.year := todaysdate.year+1;
   END
ELSE todaysdate.month := todaysdate.month+1
```

7.2.10 GOTO Statements

The *GOTO statement*, a simple statement, specifies that further processing is to continue at some other part of the program, namely, at the statement marked by the given label. The GOTO statement is presented last in this chapter because it is considered good programming practice to avoid GOTO statements whenever possible. Eliminating all or most GOTO statements improves the clarity and reliability of your programs.

The syntax of the GOTO statement is:

GOTO *label*

where

label is a statement label.

The *label* must have been declared in a label declaration (4.2) whose scope includes the block in which the GOTO falls, and it must appear in the label field of some statement within its declared scope.

The following restrictions apply to the placement of GOTO statements:

- A GOTO statement leading to the label that prefixes a statement S causes an error unless the GOTO statement is activated either by S or by a statement in the statement sequence (list of statements separated by semicolons) to which S immediately belongs. In other words, jumps into subordinate statements are not permitted.
- A GOTO statement may not refer to a case constant within a CASE statement.
- To avoid unreachable statements, a GOTO statement must be the last statement in a statement sequence (list of statements separated by semicolons), or else it must be followed by a labeled statement.
- A GOTO statement leading out of a procedure causes the termination of the procedure containing the GOTO and all procedures activated by the procedure containing the label. If more than one activation of the target procedure exists, the activation selected is the one containing the variables that are accessible at the GOTO statement. This is usually the most recent activation of the procedure.

Examples of GOTO statements:

```
GOTO 3500
GOTO 1
```

CHAPTER 8 PREDEFINED PROCEDURES AND FUNCTIONS

This chapter defines the predefined, or built-in, procedures and functions in Pascal-86. You may invoke these from any part of a program—the procedures by means of procedure statements, the functions by means of function designators within expressions.

You may also redefine any of these procedures or functions within your program by declaring your own routines with the same names. Within the scope of an explicit procedure or function declaration, this declaration overrides the predefined procedure or function.

The following kinds of predefined procedures and functions are provided in Pascal-86:

- Functions that operate on values of ordinal type
- Predicates or Boolean functions
- Arithmetic functions operating on integer or real values
- Transfer functions that perform numeric conversions on real values
- Procedures for allocation of dynamic variables
- Data transfer procedures for packing and unpacking of arrays
- Procedures for handling files and text files
- Procedures for input and output via microprocessor ports
- Interrupt handling procedures
- Procedures to communicate with an Intel 8087 Numeric Data Processor or its emulator

For each predefined procedure and function, this chapter gives a definition of what it does, the syntax of the procedure statement or function designator to invoke it, other pertinent discussion, and examples. In the syntax, optional blanks within the parameter lists have been omitted for readability.

NOTE

The predefined procedures and functions cannot be passed as arguments to procedural or functional parameters.

An expression is of an *integer type* if it is the same as one of the predefined types INTEGER, WORD, or LONGINT, or is a valid subrange of these types (5.3.1). An expression is of a *real type* if it is the same as one of the predefined types REAL, LONGREAL, or TEMPREAL.

8.1 Ordinal Functions

The predefined ordinal functions—ORD, LORD, WRD, CHR, PRED, and SUCC—all operate on a single value of an ordinal type and return values of an ordinal type.

8.1.1 ORD

The function ORD takes an expression of an ordinal type and returns a result of type INTEGER. Its calling syntax is:

```
ORD (ord-expr)
```

where

ord-expr is an expression of an ordinal type.

The result depends on the type of the argument *x*:

- If *x* is an INTEGER type, ORD(*x*) returns the value unchanged.
- If *x* is a WORD or LONGINT type whose value lies in the INTEGER range, ORD(*x*) returns the equivalent INTEGER value. If *x* is a WORD or LONGINT value outside the INTEGER range, an error occurs.
- If *x* is a CHAR-compatible type, ORD(*x*) returns the INTEGER value of the character's ordinal position in the character set. (In Pascal-86, this is the ASCII character set as defined in Appendix G).
- If *x* is any other ordinal type, ORD(*x*) returns the ordinal number determined by mapping host-type values onto consecutive non-negative INTEGER values, starting at zero. (Thus, ORD(FALSE) yields 0, and ORD(TRUE) yields 1.)

For example, given the type definition:

```
TYPE primarycolor = (red, yellow, blue);
```

then:

```
ORD(5) = 5
ORD('A') = 65 (* 41H *)
ORD(red) = 0
ORD(blue) = 2
```

8.1.2 LORD

The function LORD takes an expression of an ordinal type and returns a result of type LONGINT. Its calling syntax is:

```
LORD (ord-expr)
```

where

ord-expr is an expression of an ordinal type.

The result depends on the type of the argument *x*:

- If *x* is a LONGINT type, LORD(*x*) returns the value unchanged.
- If *x* is an INTEGER or WORD type, LORD(*x*) returns the equivalent LONGINT value.
- If *x* is a CHAR-compatible type, LORD(*x*) returns the LONGINT value of the character's ordinal position in the character set. (In Pascal-86, this is the ASCII character set, as defined in Appendix G.)
- If *x* is any other ordinal type, LORD(*x*) returns the ordinal number determined by mapping host-type values onto consecutive non-negative LONGINT values, starting at zero. (Thus, LORD(FALSE) yields 0, and LORD(TRUE) yields 1.)

When applied to an arithmetic expression, the LORD function causes the entire expression to be computed in LONGINT precision, due to the result-type propagation described in 7.1.4.

8.1.3 WRD

The function WRD takes an expression of an ordinal type and returns a result of type WORD. Its calling syntax is:

```
WRD (ord-expr)
```

where

ord-expr is an expression of an ordinal type.

The result depends on the type of argument *x*:

- If *x* is a WORD type, WRD(*x*) returns the value unchanged.
- If *x* is an INTEGER or LONGINT type whose value lies in the WORD range, WRD(*x*) returns the equivalent WORD value. If *x* is an INTEGER or LONGINT value outside the WORD range, an error occurs.
- If *x* is a CHAR-compatible type, WRD(*x*) returns the WORD value of the character's ordinal position in the character set. (In Pascal-86, this is the ASCII character set as defined in Appendix G.)
- If *x* is any other ordinal type, WRD(*x*) returns the ordinal number determined by mapping host-type values onto consecutive WORD values, starting at zero. (Thus, WRD(FALSE) yields 0, and WRD(TRUE) yields 1.)

8.1.4 CHR

The function CHR takes an expression of an integer type and returns the corresponding CHAR value according to the character collating sequence defined in Appendix G. If a corresponding character value does not exist, an error occurs.

Its calling syntax is:

```
CHR (int-expr)
```

where

int-expr is an expression of an integer type.

For any CHAR value, CHR(ORD(*x*)), where *x* is of type CHAR, always yields *x*.

For example, in Pascal-86:

```
CHR(25H) = '%'
CHR(38H) = '8'
CHR(99) = 'c' (* 63H *)
```

8.1.5 PRED

The function PRED takes an expression of an ordinal type and returns the value of the same type whose ordinal number is one less than that of the value of the expression, if such a value exists. In other words, it returns the predecessor of the argument in the ordinal type sequence.

Its calling syntax is:

```
PRED (ord-expr)
```

where

ord-expr is an expression of an ordinal type.

If no predecessor value exists, an error occurs.

For example, given the type definition:

```
TYPE primarycolor = (red, yellow, blue);
```

then:

```
PRED(blue) = yellow
PRED(yellow) = red
PRED(red) causes an error
PRED(TRUE) = FALSE
```

8.1.6 SUCC

The function SUCC takes an expression of an ordinal type and returns the value of the same type whose ordinal number is one greater than that of the value of the expression, if such a value exists. In other words, it returns the successor of the argument in the ordinal type sequence.

Its calling syntax is:

```
SUCC (ord-expr)
```

where

ord-expr is an expression of an ordinal type.

If no successor value exists, an error occurs.

For example, given the type definition:

```
TYPE primarycolor = (red, yellow, blue);
```

then:

```
SUCC(red) = yellow
SUCC(yellow) = blue
SUCC(blue) causes an error
SUCC(FALSE) = TRUE
```

8.1.7 Ordinal Type Transfer Functions

For every ordinal type, the compiler defines a transfer function with the same name. It may be used to map integers into values of the ordinal type. For example, if TYPE MONTH = (May, June, July), then June and MONTH (1) are semantically equivalent.

8.2 Predicates (Boolean Functions)

The predefined functions ODD, EOF, and EOLN all take a single argument and return a value of type BOOLEAN.

8.2.1 ODD

The function ODD takes an expression of an integer type and returns TRUE if it is odd, FALSE otherwise. Its calling syntax is:

```
ODD (int-expr)
```

where

int-expr is an expression of an integer type.

For example:

```
ODD(0) = FALSE
ODD(7) = TRUE
ODD(300) = FALSE
```

8.2.2 EOF

The function EOF takes a file argument; it returns TRUE if the associated buffer variable is positioned at the end of the file, FALSE otherwise. Its calling syntax is:

```
EOF [(file-var)]
```

where

file-var is a variable of a file type.

If the *file-var* argument is omitted, the standard file INPUT is assumed.

If the file is not open, EOF is undefined, and a call to EOF causes an error.

EOF is useful as an exit condition in loops, for example:

```
WHILE NOT EOF(f) DO BEGIN
   g := f↑; PUT(g); GET(f)
   END
```

8.2.3 EOLN

The function EOLN takes a text file argument; it returns TRUE if the associated buffer variable is positioned at the end of a line in the text file, FALSE otherwise. Its calling syntax is:

```
EOLN [(texfile-var)]
```

where

textfile-var is a variable of a text file type.

If the *textfile-var* argument is omitted, the standard file INPUT is assumed.

If the file is not open, EOLN is undefined, and a call to EOLN causes an error.

Like EOF, EOLN is useful as a loop exit condition, for example:

```
WHILE NOT EOLN DO BEGIN
    READ(number);
    sum := sum + number;
    END
```

8.3 Arithmetic Functions

For the following arithmetic functions, the operands and the returned result are of an integer or real type. An error occurs if the result value lies outside the range of the indicated result type. The arithmetic functions include ABS, SQR, SQRT, EXP, LN, SIN, COS, TAN, ARCSIN, ARCCOS, and ARCTAN.

8.3.1 ABS

The function ABS computes the absolute value of an integer or real argument. The result type is the same as the argument type.

Its calling syntax is:

ABS (*arith-expr*)

where

arith-expr is an expression of an integer or real type.

For example:

```
ABS(-5) = 5
ABS(3.777) = 3.777
ABS(0) = 0
ABS(-78000) = 78000
```

8.3.2 SQR

The function SQR computes the square of an integer or real argument. The result type is the same as the argument type.

Its calling syntax is:

SQR (*arith-expr*)

where

arith-expr is an expression of an integer or real type.

For example:

```
SQR(-5) = 25
SQR(1.2) = 1.44
SQR(0.0) = 0.0
SQR(-41000) = 1681000000
```

8.3.3 SQRT

The function SQRT computes the square root of a non-negative integer or real argument. The argument is converted to TEMPREAL format before computation, and the result is always TEMPREAL.

Its calling syntax is:

```
SQRT (arith-expr)
```

where

arith-expr is an expression of an integer or real type.

An error occurs if *arith-expr* is negative.

For example:

```
SQRT(25) = 5.0
SQRT(1.44) = 1.2
SQRT(0.0) = 0.0
SQRT(1681000000) = 41000.0
```

8.3.4 EXP

The function EXP takes an integer or real argument *x* and computes the base of natural logarithms raised to the power *x*—that is, e^x. The argument is converted to TEMPREAL format before computation, and the result is always TEMPREAL.

Its calling syntax is:

```
EXP (arith-expr)
```

where

arith-expr is an expression of an integer or real type.

For example:

```
EXP(0) = 1.0
EXP(1) = 2.7183 (* approximately *)
EXP(-1.0) = 0.367879 (* approximately *)
```

8.3.5 LN

The function LN computes the natural logarithm of a positive (greater than zero) integer or real argument. The argument is converted to TEMPREAL format before computation, and the result is always TEMPREAL.

Its calling syntax is:

```
LN (arith-expr)
```

where

arith-expr is an expression of an integer or real type.

An error occurs if *arith-expr* is not greater than zero.

For example:

```
LN(1) = 0.0
LN(2.7183) = 1.0 (* approximately *)
LN(0.367879) = -1.0 (* approximately *)
```

8.3.6 SIN

The function SIN computes the sine of an integer or real argument *x*, where *x* is in radians. The argument is converted to TEMPREAL format before computation, and the result is always TEMPREAL.

Its calling syntax is:

SIN (*arith-expr*)

where

arith-expr is an expression of an integer or real type.

For example:

```
SIN(0) = 0.0
SIN(3.1416) = 0.0 (* approximately *)
SIN(1.5708) = 1.0 (* approximately *)
SIN(-1.0) = -0.84147 (* approximately *)
```

8.3.7 COS

The function COS computes the cosine of an integer or real argument *x*, where *x* is in radians. The argument is converted to TEMPREAL format before computation, and the result is always TEMPREAL.

Its calling syntax is:

COS (*arith-expr*)

where

arith-expr is an expression of an integer or real type.

For example:

```
COS(0) = 1.0
COS(3.1416) = -1.0 (* approximately *)
COS(1.5708) = 0.0 (* approximately *)
COS(-1.0) = 0.54030 (* approximately *)
```

8.3.8 TAN

The function TAN computes the tangent of an integer or real argument *x*, where *x* is in radians. The argument is converted to TEMPREAL format before computation, and the result is always TEMPREAL.

Its calling syntax is:

```
TAN (arith-expr)
```

where

arith-expr is an expression of an integer or real type.

For example:

```
TAN(0) = 0.0
TAN(3.1416) = 0.0 (* approximately *)
TAN(0.7854) = 1.0 (* approximately *)
TAN(-1.0) = -1.5574 (* approximately *)
```

8.3.9 ARCSIN

The function ARCSIN computes the principal value, in radians, of the arcsine of an integer or real argument. The argument is converted to TEMPREAL format before computation, and the result is always TEMPREAL.

Its calling syntax is:

```
ARCSIN (arith-expr)
```

where

arith-expr is an expression of an integer or a real type.

If the absolute value of *arith-expr* is greater than 1, an error occurs.

For example:

```
ARCSIN(0) = 0.0
ARCSIN(1) = 1.5708 (* approximately* )
ARCSIN(-0.84147) = -1.0 (* approximately *)
```

8.3.10 ARCCOS

The function ARCCOS computes the principal value, in radians, of the arccosine of an integer or real argument. The argument is converted to TEMPREAL format before computation, and the result is always TEMPREAL

Its calling syntax is:

```
ARCCOS (arith-expr)
```

where

arith-expr is an expression of an integer or real type.

If the absolute value of *arith-expr* is greater than 1, an error occurs.

For example:

```
ARCCOS(1) = 0.0
ARCCOS(-1) = 3.1416 (* approximately *)
ARCCOS(0) = 1.5708 (* approximately *)
ARCCOS(0.54030) = 1.0 (* approximately *)
```

8.3.11 ARCTAN

The function ARCTAN computes the principal value, in radians, of the arctangent of an integer or real argument. The argument is converted to TEMPREAL format before computation, and the result is always TEMPREAL.

Its calling syntax is:

```
ARCTAN (arith-expr)
```

where

arith-expr is an expression of an integer or real type.

For example:

```
ARCTAN(0) = 0.0
ARCTAN(1) = 0.7854 (* approximately *)
ARCTAN(-1.5574) = -1.0 (* approximately *)
```

8.4 Transfer Functions

The transfer functions TRUNC, LTRUNC, ROUND, and LROUND perform numeric conversions. They take a single argument of a real type.

8.4.1 TRUNC

The function TRUNC takes a real expression and returns an INTEGER result that is the integer part of the real argument. The absolute value of the result is never greater than the absolute value of the argument. An invalid error occurs if the result would be outside the range of INTEGER values.

The calling syntax is:

```
TRUNC (real-expr)
```

where

real-expr is an expression of a real type.

For example:

```
TRUNC(3.7) = 3
TRUNC(-3.7) = -3
```

8.4.2 LTRUNC

The function LTRUNC takes a real expression and returns a LONGINT result that is the integer part of the real argument. The absolute value of the result is never greater than the absolute value of the argument. An invalid error occurs if the result would be outside the range of LONGINT values.

The calling syntax is:

```
LTRUNC (real-expr)
```

where

real-expr is an expression of a real type.

For example:

```
LTRUNC(69553.0787) = 69553
LTRUNC(-824000.9215) = -824000
```

8.4.3 ROUND

The function ROUND takes a real expression and returns an INTEGER result that is the value of the real argument rounded to the nearest integer. If the argument x is non-negative, then ROUND(x) is equivalent to TRUNC(x+0.5); otherwise, it is equal to TRUNC(x−0.5). An invalid error occurs if the result is outside the range of INTEGER values.

The calling syntax is:

```
ROUND (real-expr)
```

where

real-expr is an expression of a real type.

For example:

```
ROUND(3.2) = 3
ROUND(-3.7) = -4
```

8.4.4 LROUND

The function LROUND takes a real expression and returns a LONGINT result that is the value of the real argument rounded to the nearest integer. If the argument x is non-negative, then LROUND(x) is equivalent to LTRUNC(x+0.5); otherwise, it is equal to LTRUNC(x−0.5). An invalid error occurs if the result would be outside the range of LONGINT values.

The calling syntax is:

```
LROUND (real-expr)
```

where

real-expr is an expression of a real type.

For example:

```
LROUND(69553.0787) = 69553
LROUND(-824000.9215) = -824001
```

8.5 Dynamic Allocation Procedures

The predefined procedures NEW and DISPOSE allocate and deallocate dynamic variables. Each takes a pointer argument and may also, for variant record types, take one or more tag values as arguments.

8.5.1 NEW

The procedure NEW allocates a new dynamic variable. Its calling syntax is:

NEW (*pointer* [, *case-const*]...)

where

pointer is a pointer variable with base type T.

If one or more *case-const* arguments are present, T must be a record type with variants, and the *case-const* values correspond to consecutive tags of the type T, listed in the order of the matching tags in the type definition.

NEW allocates a new variable *v* of type T and assigns the pointer to *v* to *pointer*.

If *case-const* arguments are present, the allocated variable must be of a record type with variants. The allocated variable will have nested variants that correspond to the specified *case-consts*, in order of increased nesting of the variant parts. Any variant not specified must be at a deeper level of nesting than that of the last *case-const*. An error occurs if your program changes any variant part of the allocated variable to another variant. The Pascal-86 compiler does not detect this error.

In standard Pascal, an error is caused if an entire referenced variable created by a call to NEW with tag arguments is used as an operand in an expression, or as the variable on the left side of an assignment statement, or as an argument. (Such a variable does not possess the full properties—i.e., all fields—of variables of its declared record type.) However, you may reference individual components of such variables. Pascal-86 does not check for this error.

Sample program 8 in Chapter 9 illustrates the use of NEW.

8.5.2 DISPOSE

The procedure DISPOSE releases the storage allocated to a dynamic variable. Its calling syntax is:

DISPOSE (*pointer* [, *case-const*]...)

where *pointer* is a pointer variable with base type T, and the *case-const* values are tags of T as for NEW.

DISPOSE indicates that storage occupied by the variable *pointer*↑ is no longer needed. All pointer values that referenced this variable become undefined.

If you used *case-const* arguments in the call to NEW that allocated the variable, you must use at least as many *case-const* arguments in the call to DISPOSE.

An error is caused if the value of the pointer parameter is NIL or undefined, or if the pointer parameter refers to a variable that is currently a variable parameter or an element of the record variable list of an active WITH statement. Pascal-86 does not check for this error.

8.6 Transfer Procedures

The transfer procedures PACK and UNPACK provide efficient packing and unpacking of array data.

8.6.1 PACK

The procedure PACK assigns, to all elements of a packed array, corresponding elements of an unpacked array. Its calling syntax is:

PACK (*unpacked-array*, *ord-expr*, *packed-array*)

where

unpacked-array is the unpacked array variable.

packed-array is the packed array variable.

ord-expr is an expression compatible with the index type of the unpacked array.

The two array variables must have assignment-compatible component types. Furthermore, the number of components in *packed-array* must be less than or equal to the number of components in *unpacked-array*.

Let the variables **a** and **z** be declared by:

```
a: ARRAY [m..n] OF T
z: PACKED ARRAY [u..v] OF T
```

where the following relations hold:

```
ORD(n)-ORD(m) >= ORD(v)-ORD(u)
ORD(m) <= ORD(i) <= (ORD(n)-ORD(v)+ORD(u))
```

Then the statement:

```
PACK(a,i,z)
```

means:

```
k := i;
FOR j:=u TO v DO BEGIN
    z[j] := a[k];
    k := SUCC(k);
    END
```

where **j** and **k** denote auxiliary variables of appropriate type not occurring elsewhere in the program.

Note that PACK is defined for one-dimensional arrays only. To pack multidimensional arrays (arrays within arrays), you must use a loop.

8.6.2 UNPACK

The procedure UNPACK assigns, to all elements of an unpacked array, corresponding elements of a packed array. Its calling syntax is:

UNPACK (*packed-array*, *unpacked-array*, *ord-expr*)

where

packed-array is the packed array variable.

unpacked-array is the unpacked array variable.

ord-expr is an expression compatible with the index type of the packed array.

The two array variables must have assignment-compatible component types. Furthermore, the number of components in *packed-array* must be less than or equal to the number of components in *unpacked-array.*

Let the variables **a** and **z** be declared by:

```
a: ARRAY [m..n] OF T
z: PACKED ARRAY [u..v] OF T
```

where the following relations hold:

```
ORD(n)-ORD(m) >= ORD(v)-ORD(u)
ORD(m) <= ORD(i) <= (ORD(n)-ORD(v)+ORD(u))
```

Then the statement:

```
UNPACK(z,a,i)
```

means:

```
k := i;
FOR j:=u TO v DO BEGIN
  a[k] := z[j];
   k := SUCC(k);
   END
```

where **j** and **k** denote auxiliary variables of appropriate type not occurring elsewhere in the program.

Note that UNPACK is defined for one-dimensional arrays only. To unpack multidimensional arrays (arrays within arrays), you must use a loop.

8.7 File and Text File Input and Output Procedures

You can perform the following input and output operations on files, which are variables of file types as defined in 5.3.2.

- Open a file for input (RESET) or output (REWRITE)
- Read (GET) or write (PUT) one file component

- Read (READ) or write (WRITE) a sequence of file components, with automatic conversion of numbers from or to type CHAR if the file is a text file
- For text files, read a sequence of characters from a line and skip to the next line (READLN)
- For text files, write a line (WRITELN)
- For text files, write a form feed character to start a new page of printed output (PAGE)
- Check for the end of the file (EOF)
- For text files, check for the end of the current line (EOLN)

This section defines the procedures RESET, REWRITE, GET, PUT, READ, WRITE, READLN, WRITELN, and PAGE. Sections 8.2.2 and 8.2.3 define the Boolean functions EOF and EOLN.

In standard Pascal there is no procedure for closing a file. Files are closed automatically when execution returns from the program block. In Pascal-86, files may be closed with a procedure from the Run-Time Library. See Appendix B.

The predefined procedures RESET and REWRITE correspond to rewinding a tape and preparing to read from it or write to it, respectively. The procedures GET and PUT perform reading and writing, respectively, of a single file component, and also advance the position of the file. The procedures READ and WRITE transfer a sequence of file components. The predefined function EOF checks for the end of the file.

RESET positions an input file at its beginning in preparation for reading from it, and also transfers information from the first component of the file into the buffer variable; your program may then copy the value of the buffer variable into a program variable. GET advances the current file position to the next component, then assigns it. READ copies the buffer variable into a program variable you specify as a parameter, then performs a GET to fetch the next file component, repeating until the list of variables is exhausted.

REWRITE positions an output file at its beginning, in preparation for writing; it also destroys any existing information in the file. To write a new component to the file, you first assign the desired value to the file's buffer variable, then call PUT. The call to PUT appends the value of the buffer variable to the file, then causes the buffer variable's value to become undefined. WRITE copies into the buffer variable the value of an expression you specify as a parameter, then performs a PUT operation, repeating until the list of variables is exhausted.

You must use RESET before reading from, and REWRITE before writing to, any file except the predefined files INPUT and OUTPUT. In addition, you may not follow a write operation (PUT, WRITE, or WRITELN) by a read (GET, READ, or READLN), or a read by a write; so if you are using the same file for both input and output, you must rewind it with REWRITE, write to it, rewind it with RESET, then read from it, and so on.

In standard Pascal, an error is caused if you alter the position of a file while the buffer variable F↑ is either an argument to a variable parameter or an element of the record variable list of an active WITH statement or both. However, the Pascal-86 compiler does not check for this non-standard usage.

With non-text files declared as FILE OF CHAR, the procedures GET and READ read in the characters, one at a time, just as they appear in the file. With text files, however, GET, READ, and READLN interpret end-of-line markers as single blanks and ignore form feed characters.

The action of the standard Pascal input buffering scheme, as described in the preceding explanation of RESET, GET, and READ, is to read ahead in the file so that input may be overlapped with computations for efficiency. However, this scheme is not appropriate for interactive programs, since it may allow a program to query the terminal for input before the terminal has prompted the user for that input. For this reason, the Pascal-86 run-time system uses an alternative scheme, called lazy input, for files declared as type TEXT or FILE OF CHAR (but not PACKED FILE OF CHAR, for which the run-time system uses the standard buffering scheme).

Under *lazy input*, a "buffer full" flag associated with the file controls the filling of the buffer variable. RESET and GET always clear the flag to indicate that new data is needed. Whenever the buffer variable is referenced, the flag is tested, and if it is cleared, the buffer variable is filled from the file (e.g., the terminal). After the buffer variable has been filled, the "buffer full" flag is set, so that subsequent references to the buffer variable (without intervening calls to GET) will not initiate another read from the file. Invocations of EOF and EOLN make sure the buffer is full before testing for end of file or end of line.

Note that the calling sequences of READ, WRITE, READLN, and WRITELN differ from those of other predefined and user-defined procedures and functions in Pascal. Each of these procedures takes a variable number of parameters, and the types of the parameters are not predetermined by the procedure.

8.7.1 RESET

The procedure RESET opens a file for input. It positions the file at its beginning in preparation for reading from it, and also transfers information from the first component of the file into the buffer variable. Your program may then copy the value of the buffer variable into a program variable. You must use RESET before using GET, READ, or READLN to read from any file except the predefined file INPUT.

Its calling syntax is:

RESET (*file-var*, *string-expr*)

where *file-var* is the name of a file variable, and the optional *string-expr* is a stringtype (PACKED ARRAY [1..n] OF CHAR) expression that specifies a physical file to be associated with the *file-var*. Note that *string-expr* cannot be a CHAR variable or string constant of length one; it must be either a variable with a stringtype, or a string constant with two or more characters. Because stringtypes are of fixed length and filenames are often variable length, the variable of stringtype may use an ASCII nul (0) to mark the end of the string.

RESET resets the current file position at its beginning. If the specified file f is not empty, RESET assigns the buffer variable f ↑ to the value of the first component of f, and causes EOF(f) to become FALSE. If f is empty or does not exist, f ↑ becomes undefined, and EOF(f) becomes TRUE.

The *string-expr*, if present, must be in a form acceptable to the operating system under which you run your programs, and can be at most 45 characters long. If a *string-expr* is given, the specified physical file name overrides any preconnection for that file variable. (Preconnection is described in 12.5.1.)

If no *string-expr* is given, the physical file last associated with the file variable is assumed. If the first reference to a file variable has no physical file associated with it (either by preconnection or by a *string-expr* parameter to RESET or REWRITE), a default physical file is assigned to it. If the file variable is a program parameter specified in the program heading, the default physical file name is the same as the file variable name. If the file variable is not a program parameter, the run-time system creates an empty temporary file, which will be deleted at the end of the program.

Sample programs 2A, 2B, 3, and 4 in Chapter 9 give examples of the use of RESET.

8.7.2 REWRITE

The procedure REWRITE opens a file for output. It positions a file at its beginning in preparation for writing to it and also destroys any existing information in the file. You must use REWRITE before using PUT, WRITE, or WRITELN to write to any file except the predefined file OUTPUT.

Its calling syntax is:

REWRITE (*file-var*, *string-expr*)

where *file-var* is the name of a file variable, and the optional *string-expr* is a stringtype (PACKED ARRAY [1..n] OF CHAR) expression that specifies a physical file to be associated with the *file-var*. Note that *string-expr* cannot be a CHAR variable or string constant of length one; it must be either a variable with a stringtype, or a string constant with two or more characters. Because stringtypes are of fixed length and filenames are often variable length, the variable of stringtype may use an ASCII nul (0) to mark the end of the string.

REWRITE positions the specified file f such that a new file may be generated, causes EOF to become TRUE, and causes the buffer variable f ↑ to become undefined.

The *string-expr*, if present, must be in a form acceptable to the operating system under which you run your programs, and can be at most 45 characters long. If a *string-expr* is given, the specified physical file name overrides any preconnection for that file variable. (Preconnection is described in 12.5.1.)

If no *string-expr* is given, the physical file last associated with the file variable is assumed. If the first reference to a file variable has no physical file associated with it (either by preconnection or by a *string-expr* parameter to RESET or REWRITE), a default physical file is assigned to it. If the file variable is a program parameter specified in the program heading, the default physical file name is the same as the file variable name. If the file variable is not a program parameter, the run-time system creates a temporary file, which will be deleted at the end of the program.

If the file already exists, REWRITE will delete it and create an empty new file. In essence, it will erase the contents of the file.

Sample program 3 in Chapter 9 gives an example of the use of REWRITE.

8.7.3 GET

The procedure GET advances the current file position to the next component, then assigns the value of this component to the associated buffer variable, allowing your program to copy it.

Its calling syntax is:

GET [(*file-var*)]

where

file-var	is the name of a file variable.

If a file variable is not specified, the predefined text file INPUT is used.

If the predicate EOF(f) is FALSE prior to the execution of GET(f), then GET advances the current file position to the next component and assigns the value of this component to the buffer variable f↑. If no next component exists, then EOF(f) becomes TRUE, and the value of f↑ becomes undefined. If EOF(f) is TRUE prior to execution, an error occurs.

When GET is applied to a file f, an error occurs if f is undefined. An error also occurs if the file is set for output rather than input; that is, if REWRITE(f) has been called since the last call to RESET(f).

Sample program 3 in Chapter 9 gives an example of the use of GET.

8.7.4 PUT

The procedure PUT appends the value of the buffer variable to the specified file, then causes the value of the buffer variable to become undefined.

Its calling syntax is:

PUT [(*file-var*)]

where

file-var is the name of a file variable.

If a file variable is not specified, the predefined text file OUTPUT is used.

If the predicate EOF(f) is TRUE prior to the execution of PUT(f), then PUT appends the buffer variable f↑ to the file f, EOF(f) remains TRUE, and the value of f↑ becomes undefined. If EOF(f) is FALSE prior to execution, an error occurs.

When PUT is applied to a file f, an error occurs if f is undefined. An error also occurs if the file is set for input rather than output; that is, if RESET(f) has been called since the last call to REWRITE(f).

Sample program 3 in Chapter 9 gives an example of the use of PUT.

8.7.5 READ

The READ procedure copies the file buffer variable into a program variable you specify as an argument, then performs a GET to fetch the next file component, repeating until the list of variables is exhausted. The file must be a text file; READ automatically converts the input from type CHAR to the appropriate variable type(s).

Its calling syntax is:

READ ([*file-var*,]*variable*[, *variable*]...)

where

file-var is the name of a file variable.

If *file-var* is omitted, the predefined text file INPUT is assumed. Each *variable* parameter may be an entire variable, a component of a structured variable (indexed variable, field designator, or buffer variable) or a referenced variable.

When READ is applied to a file *f*, an error occurs if *f* is undefined. An error also occurs if the file is set for output rather than input; that is, if REWRITE(f) has been called since the last call to RESET(f).

Sample programs 2A, 2B, 4, 6, 7, and 8 in Chapter 9 give examples of the use of READ.

Each variable must be of a character, integer, or real type, or a valid subrange of these types as defined in 5.3.1.

The statement:

```
READ(f,v1,...,vn)
```

is equivalent to:

```
BEGIN READ(f,v1); ... ; READ(f,vn) END
```

Character Variables. If **v** is a variable of type CHAR or a subrange of CHAR, then:

```
READ(f,v)
```

is equivalent to:

```
BEGIN v:=f↑; GET(f) END
```

Integer Variables. If **v** is a variable of type INTEGER, WORD, or LONGINT, or a valid subrange of these types, then:

```
READ(f,v)
```

reads from file **f** a sequence of characters that form a decimal integer; that is, a sequence of decimal digits. The value of the integer, which must be assignment-compatible with the type of **v**, is assigned to **v**. Preceding spaces and end-of-line markers are skipped. Reading ceases as soon as the buffer variable f↑ contains a character that does not form part of the integer. If the first character read is not a legal decimal digit or a minus sign followed by a legal decimal digit, an error occurs.

NOTE

> READ will not skip characters other than a space, carriage return, or line feed. In particular, the comma (,) is not skipped and will cause an error if used as a delimiter.

Real Variables. If **v** is a variable of type REAL, LONGREAL, or TEMPREAL, then:

```
READ(f,v)
```

reads from file **f** a sequence of characters that represent a real number.

In standard Pascal, the characters read must be legal real numbers, as defined in 3.3.3; in other words, they must follow the syntax rules for compile-time real constants.

In Pascal-86, the syntax for real input from a text file is less restrictive. A real number recognized by READ or READLN in Pascal-86 may have any of the following five forms:

```
[sign][digits][ . ]digits[ E [sign]digits]
[sign]digits[ . ][digits][ E [sign]digits]
+[ + ]...
-[ - ]...
.[ . ]...
```

where

sign is a plus or a minus sign.

digits is a sequence of one or more decimal digits.

A string of two or more plus signs is interpreted as plus infinity. A string of two or more minus signs is interpreted as minus infinity. A string of two or more periods is interpreted as the NaN that represents an indefinite value.

A maximum of 20 significant digits will be interpreted. If more than 20 digits are given, the least significant digits will be ignored.

The value of the real number is assigned to **v**. Preceding spaces and end-of-line markers are skipped. Reading ceases as soon as the buffer variable f ↑ contains a character that does not form part of the real number. If the sequence of characters read does not form a legal real number, an error occurs.

NOTE

You cannot apply READ to nontext files. You must use GET(f).

8.7.6 WRITE

The WRITE procedure copies into the file buffer variable the value of an expression you specify as an argument, then performs a PUT, repeating until the list of arguments is exhausted. The file must be a text file; WRITE automatically converts all arguments to type CHAR before writing them.

Its calling syntax is:

WRITE ([*file-var*,]*write-pararm*[, *write-param*]...)

where

write-param is of the form:

expression [: *total-width-expn* [: *frac-digits-expn*]]

The *file-var* is the name of file variable. If the *file-var* is omitted, the predefined text file OUTPUT is assumed. Each *expression* is an expression whose value will be written to the file *f*; it may be of a character, integer, real, Boolean, or string type. The items *total-width-expn* and *frac-digits-expn* are expressions of an integer type; their values must be greater than or equal to 1, or an error occurs.

The statement:

```
WRITE(f,p1,...,pn)
```

(where p1,...,pn are *write-params*) is equivalent to:

```
BEGIN WRITE(f,p1); ... ; WRITE(f,pn) END
```

When WRITE is applied to a file *f*, an error occurs if *f* is undefined. An error also occurs if the file is set for input rather than output; that is, if RESET(f) has been called since the last call to REWRITE(f).

Sample programs 1, 2A, 2B, 3, 4, 6, and 7 in Chapter 9 give examples of the use of WRITE.

WRITE for Text Files

In each *write-param* in the calling syntax, *expression* is the value to be written, and *total-width-expn* and *frac-digits-expn* are the field width parameters. The value of *total-width-expn* is the total number of characters to be written, unless *expression* requires more than *total-width-expn* characters to represent it; in this case, the number of characters written will be the smallest number necessary to represent *expression.* If *total-width-expn* is omitted, a default total width value is assumed. In Pascal-86, this default value is 1 for character, integer, real, and Boolean expressions, and n for string expressions (the number of characters specified for the string type).

The field width parameter *frac-digits-expn* may be used only when *expression* is of a real type. Its presence specifies output in fixed-point representation, and its absence specifies floating-point representation. For details, see the discussion of real output later in this section.

Character Expressions. If the expression to be written is of type CHAR, the value written is right-adjusted in an output field of the specified width, and any remaining positions to the left are filled with blanks.

Integer Expressions. If the expression to be written is of type INTEGER, WORD, or LONGINT, the decimal representation of its value will be written, preceded by a minus sign if the number is negative. The representation of zero is a single zero digit. If the specified total width of the output field is large enough to contain the decimal representation (and the minus sign if the number is negative), the value written is right-adjusted in an output field of the specified width, and any remaining positions to the left are filled with blanks. Otherwise, the value is written using an output field of as many characters as needed.

Real Expressions. If the expression to be written is an expression of a real type, a decimal representation of the value, rounded to the specified number of significant digits or decimal places, is written. If *frac-digits-expn* is given, the number is written in fixed-point representation; otherwise, it is written in floating-point representation.

In Pascal-86, floating-point arithmetic may produce negative zeros, positive and negative infinities, and NaN's. WRITE represents these as follows:

- Negative zero is shown as a zero digit preceded by a minus sign.
- Positive infinity is shown as a string of plus signs (+) that fill the specified field.
- Negative infinity is shown as a string of minus signs (−) that fill the specified field.
- A NaN (Not a Number) is shown as a string of dots (.) that fill the specified field.

Floating-Point Representation. If a real expression is to be written and *frac-digits-expn* is not given, the value of the expression is written in floating-point representation.

This is a decimal representation in scientific notation form, with one digit to the left of the decimal point, truncated to fit the actual width of the output field. If the value of *total-width-expn* is greater than or equal to 10, the actual width of the output field is as specified by *total-width-expn*; otherwise, the actual width is 10.

The floating-point representation has the following form:

1. The sign of the real value (— or blank)
2. The most significant digit of the scientific notation representation (0 if the value is zero)
3. The decimal point (.)
4. The next *f* significant digits of the scientific notation representation, where f = actual width - 9 (all zeros if the value is zero)
5. A capital E
6. The sign of the exponent (— or +)
7. The exponent in four digits, with leading zeros if necessary

Note that making *total-width-expn* larger will only increase the number of significant digits written out. If you desire padding blanks, your program must write them out explicitly.

Fixed-Point Representation. If a real expression is to be written and *frac-digits-expn* is given, the value of the expression is written in fixed-point representation.

Like the floating-point representation, this is a decimal representation. The number of digits to the left of the decimal point (*i*) is the number of digits needed to represent the integer part of the value, and the number of digits to the right of the decimal point (*f*) is specified in *frac-digits-expn*. If the specified length of the significand is less than or equal to 18 digits, the number will be rounded to fit the output field. Otherwise, the number will be truncated to 18 significant digits, adding trailing zeros if needed.

The minimum number of characters written (*m*) is thus (i+f+1) for a positive or zero value, or (i+f+2) for a negative value. If the value of *total-width-expn* is greater than or equal to *m*, the output value is right-adjusted in an output field of the specified width, and any remaining positions to the left are filled with blanks. Otherwise, the value is written in an output field of width *m*.

NOTE

The fixed-point equivalent of a real expression may have over 4000 decimal places. For this reason, your program should check the ranges of real values before printing them out in fixed-point form.

Following the leading blanks if any, the fixed-point representation has the following form:

1. A minus sign (—) if the value is negative
2. The first *i* digits of the value
3. The decimal point (.)
4. The next *f* digits of the value

String Expressions. If the expression to be written is of a string type, the value of the string will be written as a sequence of characters. If the specified width of the output field is greater than or equal to the length of the string, the string is rightadjusted in an output field of the specified width, and any remaining positions to the left are filled with blanks. Otherwise, the string is truncated on the right to the widths of the output field before it is written.

NOTE

You must use PUT(f) for writing nontext files.

8.7.7 READLN

The READLN procedure is applicable only to text files. It performs the same functions as READ, but after it has read in all the specified data, it moves the position of the file to just past the end of the current line. Unless this is the end-of-file position, READLN thus positions the file at the start of the next line, skipping over the last part of the line just read. You may call READLN without specifying any variables. In this case, it simply moves the position of the file.

Its calling syntax may be any one of the following:

```
READLN ([textfile-var, ]variable[, variable]...)
READLN (text-file)
READLN
```

where

textfile-var is the name of a text file variable.

If *textfile-var* is not present, the predefined text file INPUT is assumed. Each *variable* parameter may be an entire variable, a component of a structured variable (indexed variable, field designator, or buffer variable) or a referenced variable. Furthermore, each *variable* must be of a character, integer, or real type, or a valild subrange of these types as defined in 5.3.1.

The statement:

```
READLN(f,v1,..,  vn)
```

is equivalent to:

```
BEGIN READ(f,v1,...,vn); READLN(f) END
```

where:

```
READLN(f)
```

is equivalent to:

```
BEGIN WHILE NOT EOLN(f) DO GET(f); GET(f) END
```

When READLN is applied to a text file *f*, an error occurs if *f* is undefined. An error also occurs if the file is set for output rather than input; that is, if REWRITE(f) has been called since the last call to RESET(f).

Sample programs 1, 2A, 2B, 4, 6, 7, and 8 in Chapter 9 give examples of the use of READLN.

8.7.8 WRITELN

The WRITELN procedure is applicable only to text files. It performs the same functions as WRITE, but after it has written all the specified data, it writes an end-of-line marker (in the Pascal-86 logical record system, an ASCII carriage return followed by a line feed), thus terminating the current line. You may call WRITELN without specifying any variables; in this case, it simply writes an end-of-line marker (producing a skipped line on printed output).

Its calling syntax may be any one of the following:

```
WRITELN ([textfile-var, ]write-param[ , write-param]...)
WRITELN (textfile-var)
WRITELN
```

where

write-param is of the form:

expression [: *total-width-expn* [: *trac-digits-expn*]]

The *textfile-var* is the name of a text file variable. If *textfile-var* is not present, the predefined text file OUTPUT is assumed. Each *expression* is an expression whose value is to be written to the file f; this expression may be of a character, integer, real, Boolean, or string type. The items *total-width-expn* and *frac-digits-expn* are expressions of type INTEGER. Their values must be greater than or equal to 1, or an error occurs. The *total-width-expn* and *frac-digits-expn* parts are explained further in the section on WRITE for text files.

The statement:

```
WRITELN(f,p1,...,pn)
```

is equivalent to:

```
BEGIN WRITE(f,p1,...,pn); WRITELN(f) END
```

When WRITELN is applied to a text file *f*, an error occurs if *f* is undefined. An error also occurs if the file is set for input rather than output; that is, if RESET(f) has been called since the last call to REWRITE(f).

In standard Pascal, WRITELN is the only means for writing the line marker. In Pascal-86, WRITEing CR followed by LF has the same effect.

Sample programs 1, 2A, 2B, 3, 4, 6 ,7, and 8 in Chapter 9 give examples of the use of WRITELN.

8.7.9 PAGE

The procedure PAGE is applicable only to text files. It writes a form feed character (in Pascal-86, ASCII 0CH) to the file. This causes a page eject in the file when it is printed, so that subsequent output to the file will be on a new page. If the last character sequence written before the call to PAGE was not an end-of-line marker (CR followed by LF for Pascal-86), PAGE performs an implicit WRITELN to the file.

Its calling syntax is:

```
PAGE [(textfile-var)]
```

where

textfile-var is a text file variable.

If *textfile-var* is omitted, the predefined text file variable OUTPUT is assumed.

When PAGE is applied to a text file *f*, an error occurs if *f* is undefined. An error also occurs if the file is set for input rather than output; that is, if RESET(f) has been called since the last call to REWRITE(f).

For example:

```
PAGE(edittext)
```

causes a page eject in the file **edittext**.

8.8 Port Input and Output Procedures

The port input and output procedures provide access to the I/O ports of the 8086 or 8088 CPU. INBYT and OUTBYT transfer a single byte of data; INWRD and OUTWRD transfer two consecutive bytes. No formatting is done. These procedures provide fast data transfer by means of direct communication with the microprocessor hardware. Sample program 5 in Chapter 9 illustrates their use.

You must avoid using I/O ports that are reserved for your operating system.

8.8.1 INBYT

The procedure INBYT transfers a single byte of data from an 8086 or 8088 input port. Its calling syntax is:

```
INBYT (port-address, variable)
```

where

port-address is an integer expression in the range −32767 to 65535, inclusive.

variable is a variable of any type that occupies a single byte.

The value of *port-address* designates the input port number (65536 + *port-address* for negative values), and *variable* designates the variable to receive the data.

For example, if **nextchar** is a variable of type CHAR, then:

```
INBYT(5,nextchar)
```

transfers one byte from input port 5 to the variable **nextchar**.

8.8.2 INWRD

The procedure INWRD transfers two consecutive bytes of data from an 8086 or 8088 input port. Its calling syntax is:

```
INWRD (port-address, variable)
```

where

port-address is an integer expression in the range −32767 to 65535, inclusive.

variable is a variable of any type that occupies a word (two consecutive bytes).

The value of *port-address* designates the input port number (65536 + *port-address* for negative values), and *variable* designates the variable to receive the data.

For example, if **int** is a variable of type INTEGER, then:

```
INWRD(18,int)
```

transfers one word from input port 18 to the variable int.

8.8.3 OUTBYT

The procedure OUTBYT transfers a single byte of data to an 8086 or 8088 output port. Its calling syntax is:

```
OUTBYT (port-address, expression)
```

where

port-address is an integer expression in the range −32767 to 65535, inclusive.

expression is an expression of any type that occupies a single byte.

The value of *port-address* designates the output port number (65536 +*port-address* for negative values), and *expression* designates the value to be output.

For example, if **switch1** and **switch2** are variables of type BOOLEAN, then:

```
OUTBYT (1, switch1 AND switch2)
```

transfers the value of the Boolean expression **switch1 AND switch2** to output port 1.

8.8.4 OUTWRD

The procedure OUTWRD transfers two consecutive bytes of data to an 8086 or 8088 output port. Its calling syntax is:

```
OUTWRD (port-address, expression)
```

where

port-address is an integer expression in the range −32767 to 65535, inclusive.

expression is an expression of any type that occupies a word (two consecutive bytes).

The value of *port-address* designates the output port number (65536 + *port-address* for negative values), and *expression* designates the value to be output.

For example, if **int** is a variable of type INTEGER, then:

```
OUTWRD(20, int+1)
```

transfers the value of the INTEGER expression **int+1** to output port 20.

8.9 Interrupt Control Procedures

Four procedures are provided to aid interrupt processing on the 8086 or 8088 CPU: SETINTERRUPT, ENABLEINTERRUPTS, DISABLEINTERRUPTS, and CAUSEINTERRUPT. Sample program 5 in Chapter 9 illustrates the use of these procedures.

The 8086 and 8088 interrupt handling mechanism has two states: enabled and disabled. At the start of a Pascal-86 program, interrupts are enabled.

You must avoid using interrupt levels that are reserved for the Pascal-86 run-time system or (if your program performs real arithmetic) the interrupt level reserved for the 8087 processor or emulator. For details, see section K.1 in Appendix K. You must also avoid using interrupt levels that are reserved for your operating system.

> **NOTE**
>
> The software scheme used for interrupt processing is dependent on the operating system. For some operating systems, these procedures may conflict with the way the operating system views the interrupt model. Consequently, you may need to call the operating system directly to achieve the desired interrupt control.

8.9.1 SETINTERRUPT

The procedure SETINTERRUPT provides a way to dynamically associate an interrupt procedure (designated as such by using the INTERRUPT control described in 10.4.7) with a given interrupt number. Its calling syntax is:

```
SETINTERRUPT (int-expr, identifier)
```

where

int-expr is an integer expression in the range 0 to 255, inclusive.

identifier is the name of an interrupt procedure (see 10.4.9).

SETINTERRUPT associates the named interrupt procedure with the interrupt whose number is the value of *int-expr*.

For example:

```
SETINTERRUPT(7,overtemp)
```

where **overtemp** is an interrupt procedure, will cause **overtemp** to be associated with interrupt number 7.

8.9.2 ENABLEINTERRUPTS

The procedure ENABLEINTERRUPTS forces the 8086 or 8088 interrupt mechanism into the enabled state. Its calling syntax is:

```
ENABLEINTERRUPTS
```

8.9.3 DISABLEINTERRUPTS

The procedure DISABLEINTERRUPTS forces the 8086 or 8088 interrupt mechanism to be disabled, so that all interrupts will be ignored. Its calling syntax is:

```
DISABLEINTERRUPTS
```

8.9.4 CAUSEINTERRUPT

The procedure CAUSEINTERRUPT causes an interrupt of the specified number to occur. Its calling syntax is:

```
CAUSEINTERRUPT (int-expr)
```

where

int-expr is an integer expression in the range 0 to 255, inclusive.

CAUSEINTERRUPT causes interrupt *n* to occur, where *n* is the value of *int-expr*.

For example:

```
CAUSEINTERRUPT(33)
```

causes interrupt 33 to occur.

8.10 8087 Procedures

Pascal-86 provides two procedures to communicate with the floating-point (real) arithmetic error handling mechanism of the 8087 processor or emulator: GET8087ERRORS and MASK8087ERRORS.

It is recommended that you use these procedures to check for real arithmetic errors, as described at the end of section 7.1.8. Section 14.6 gives a fuller discussion of real arithmetic error handling.

Two predefined types, AT87ERRORS and AT87EXCEPTIONS, are provided in Pascal-86 for use with these procedures. These types are defined as follows:

```
TYPE AT87ERRORS = SET OF AT87EXCEPTIONS;
     AT87EXCEPTIONS =
          (AT87NVLD,AT87DENR,AT87ZDIV,AT87OVER,
          AT87UNDR,AT87PRCN,AT87RSVD,AT87MASK);
```

AT87NVLD stands for the invalid operation exception, AT87DENR for the denormalized operand exception, AT87ZDIV for zero divide, AT87OVER for overflow, AT87UNDR for underflow, AT87PRCN for precision, AT87MASK for the 8087 interrupt enable mask bit, and AT87RSVD for a bit in the error field that is currently reserved. The types AT87ERRORS and AT87EXCEPTIONS and the constants AT87NVLD, AT87DENR, AT87ZDIV, AT87OVER, AT87UNDR, AT87PRCN, AT87RSVD, and AT87MASK are predefined in Pascal-86. Sections 7.1.8 and 14.6 discuss these errors and the real error handling mechanism in Pascal-86.

8.10.1 GET8087ERRORS

The procedure GET8087ERRORS returns the error field of the 8087 status word, then clears the error field. Its calling syntax is:

```
GET8087ERRORS (variable)
```

where

variable is the variable in which the error field is to be returned. It is of type AT87ERRORS.

Unless you are running your program with all real errors unmasked, you should call GET8087ERRORS at the end of any significant computation sequence, to check for masked exceptions.

For example:

```
GET8087ERRORS(errors)
```

where **errors** is a variable of type AT87ERRORS as defined above, stores the value of the 8087 error mask in **errors**.

8.10.2 MASK8087ERRORS

The procedure MASK8087ERRORS sets the real error mask in the 8087 chip or emulator to a given value. Its calling syntax is:

```
MASK8087ERRORS (expression)
```

where

expression is the value to be set in the error mask. It is of type AT87ERRORS.

The initial setting of the error mask is:

```
[AT87DENR,AT87ZDIV,AT87OVER,AT87UNDR,AT87PRCN]
```

That is, all exceptions but AT87NVLD are masked, and interrupts are enabled. In a procedure or function in which the 8087 error mask is changed, the value of the error mask is saved on entry to the procedure or function, and the saved value is automatically reinstated on exit. Sections 7.1.8 and 14.6 give details on floatingpoint error handling.

For example:

```
MASK8087ERRORS([ ])
```

resets the error mask so that all errors are unmasked.

CHAPTER 9
SAMPLE PROGRAMS

This chapter includes nine sample programs illustrating important features of the Pascal language. Two of these have already been introduced in Chapters 1 and 2.

Source code for all these programs is provided on the release diskette, so that you may run them yourself. For examples of linking, locating and executing programs, see Chapter 12.

9.1 Sample Program 1: Temperature Conversion

The program in figure 9-1, introduced in Chapter 1, converts Fahrenheit degrees to Celsius as you enter them from the console.

When you run this program, it first displays the message:

```
Fahrenheit temperature is:
```

Type in a temperature in Fahrenheit degrees. If you mistype, you may edit the line using the RUBOUT key. Then type RETURN.

The program calculates the Celsius temperature and displays the output:

```
Celsius temperature is: n
```

where

n is the Celsius equivalent of the temperature you typed in.

Finally, the program skips a space and displays:

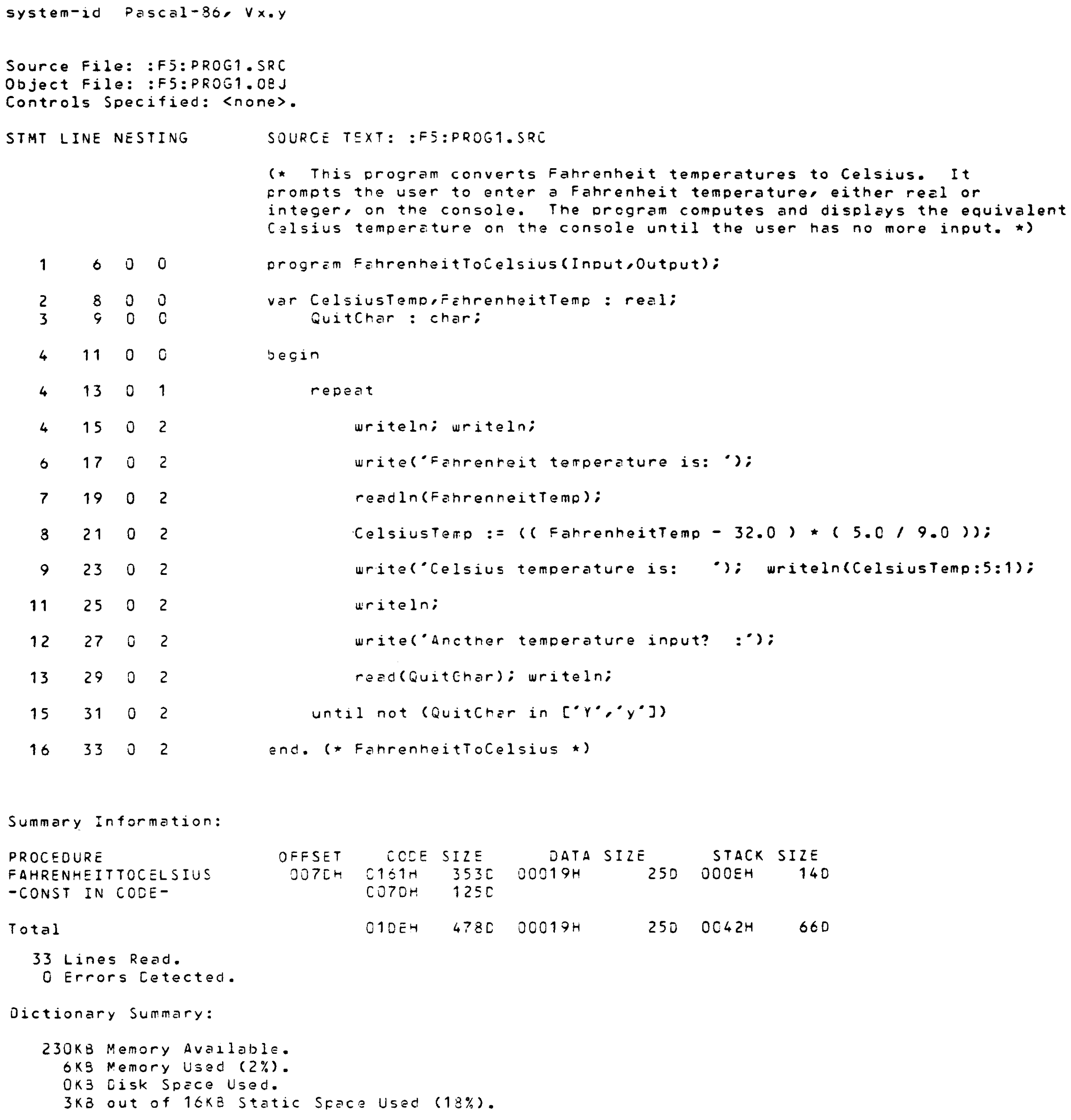
```
Another temperature input? :
```

Type Y or y if you want to do another calculation. This causes the program to skip a line and display the starting message again, allowing you to type in another temperature. You may do this as many times as you wish.

When you wish to stop, answer the final query with any character other than Y or y, and the program will skip a line and return control to the operating system. This program must be linked to the run-time support libraries (P86RN0.LIB, P86RN1.LIB, P86RN2.LIB, P86RN3.LIB), the appropriate 8087 libraries (either 8087.LIB or E8087.LIB and E8087), and any interface libraries required by your operating system.

9.2 Sample Programs 2A and 2B: Binary Tree Traversal

These programs, introduced in Chapter 2, build a binary tree and print out the nodes of the tree in three notations: infix, prefix, and postfix. These notations represent the three methods of tree traversal—inorder, preorder, and postorder, respectively—used in most computer programs that deal with trees. Such a tree may represent, for example, an arithmetic expression to be interpreted by a parsing program, with the nodes being symbols or tokens. Here, for simplicity, each node is a single character.



```
system-id  Pascal-86, Vx.y

Source File: :F5:PROG1.SRC
Object File: :F5:PROG1.OBJ
Controls Specified: <none>.

STMT LINE NESTING       SOURCE TEXT: :F5:PROG1.SRC

                        (*  This program converts Fahrenheit temperatures to Celsius.  It
                        prompts the user to enter a Fahrenheit temperature, either real or
                        integer, on the console.  The program computes and displays the equivalent
                        Celsius temperature on the console until the user has no more input. *)

  1    6  0  0          program FahrenheitToCelsius(Input,Output);

  2    8  0  0          var CelsiusTemp,FahrenheitTemp : real;
  3    9  0  0              QuitChar : char;

  4   11  0  0          begin

  4   13  0  1              repeat

  4   15  0  2                  writeln; writeln;

  6   17  0  2                  write('Fahrenheit temperature is: ');

  7   19  0  2                  readln(FahrenheitTemp);

  8   21  0  2                  CelsiusTemp := (( FahrenheitTemp - 32.0 ) * ( 5.0 / 9.0 ));

  9   23  0  2                  write('Celsius temperature is:   '); writeln(CelsiusTemp:5:1);

 11   25  0  2                  writeln;

 12   27  0  2                  write('Another temperature input?  :');

 13   29  0  2                  read(QuitChar); writeln;

 15   31  0  2              until not (QuitChar in ['Y','y'])

 16   33  0  2          end. (* FahrenheitToCelsius *)


Summary Information:

PROCEDURE                  OFFSET    CODE SIZE      DATA SIZE       STACK SIZE
FAHRENHEITTOCELSIUS         007DH  0161H   353D  00019H      25D  000EH     14D
-CONST IN CODE-                    007DH   125D

Total                              01DEH   478D  00019H      25D  0042H     66D

   33 Lines Read.
    0 Errors Detected.

Dictionary Summary:

   230KB Memory Available.
     6KB Memory Used (2%).
     0KB Disk Space Used.
     3KB out of 16KB Static Space Used (18%).
```

Figure 9-1. Sample Program 1: Temperature Conversion

The input is in the form of a series of lines, each representing a node by means of four items separated by blanks. The first item in the input line is the character at that node. The second is a sequence number identifying the node; the program uses this number as an index into an array of records representing the tree. The third and fourth items are the sequence numbers of the left-hand and right-hand nodes, respectively, that are connected below the given node. A zero means there is no node connected below the given node at that (left-hand or right-hand) position. For example:

```
+  1  2  3
*  2  4  5
E  3  0  0
-  4  6  7
D  5  0  0
/  6  8  9
C  7  0  0
A  8  0  0
B  9  0  0
```

This input represents the tree in figure 9-2.

The three methods of tree traversal are all defined recursively. The starting point (the first root) is at the top of the tree. For the three methods, the steps at each level of recursion are as follows:

Inorder traversal:
1. Traverse the left sub-tree.
2. Visit the root.
3. Traverse the right sub-tree.

Preorder traversal:
1. Visit the root.
2. Traverse the left sub-tree.
3. Traverse the right sub-tree.

Postorder traversal:
1. Traverse the left sub-tree.
2. Traverse the right sub-tree.
3. Visit the root.

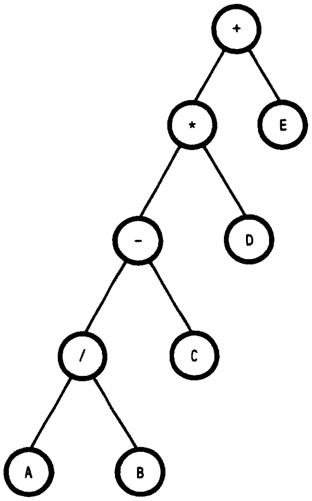


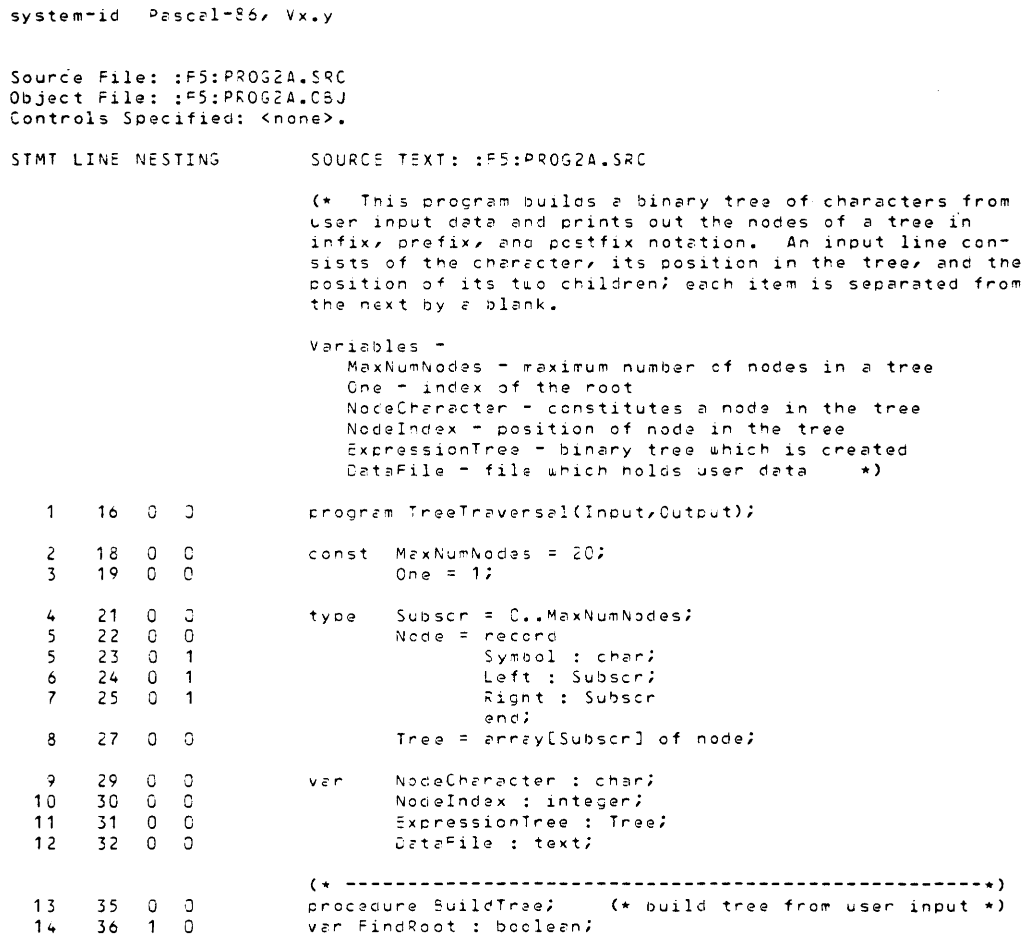
Figure 9-2. Sample Input Tree for Sample Programs 2A and 2B 121539-35

Thus, the output produced from the sample input is as follows:

```
INFIX:
 (((( A / B) - C) * D) + E)
PREFIX:
 + * - / A B C D E
POSTFIX:
A B / C - D * E +
```

These programs show the use of nested structures—in this case, an array of records. The procedures **infix**, **prefix**, and **postfix** are all directly recursive, reflecting the recursive definitions of the three methods of tree traversal.

Sample program 2A (figure 9-3) is a standard Pascal version of the tree traversal program. Sample program 2B (figure 9-4) is the source program divided into two separately compiled modules, using the separate compilation facilities of Pascal-86. The programs must be linked to the run-time support libraries (P86RN0.LIB, P86RN1.LIB, P86RN2.LIB, P86RN3.LIB), 87NULL.LIB, and any interface libraries required by your operating system.



```
system-id  Pascal-86, Vx.y

Source File: :F5:PROG2A.SRC
Object File: :F5:PROG2A.OBJ
Controls Specified: <none>.

STMT LINE NESTING        SOURCE TEXT: :F5:PROG2A.SRC

                         (* This program builds a binary tree of characters from
                         user input data and prints out the nodes of a tree in
                         infix, prefix, and postfix notation.  An input line con-
                         sists of the character, its position in the tree, and the
                         position of its two children; each item is separated from
                         the next by a blank.

                         Variables -
                            MaxNumNodes - maximum number of nodes in a tree
                            One - index of the root
                            NodeCharacter - constitutes a node in the tree
                            NodeIndex - position of node in the tree
                            ExpressionTree - binary tree which is created
                            DataFile - file which holds user data      *)

  1   16  0  0           program TreeTraversal(Input,Output);

  2   18  0  0           const  MaxNumNodes = 20;
  3   19  0  0                  One = 1;

  4   21  0  0           type   Subscr = 0..MaxNumNodes;
  5   22  0  0                  Node = record
  5   23  0  1                         Symbol : char;
  6   24  0  1                         Left : Subscr;
  7   25  0  1                         Right : Subscr
                                        end;
  8   27  0  0                  Tree = array[Subscr] of node;

  9   29  0  0           var    NodeCharacter : char;
 10   30  0  0                  NodeIndex : integer;
 11   31  0  0                  ExpressionTree : Tree;
 12   32  0  0                  DataFile : text;

                         (* ---------------------------------------------------*)
 13   35  0  0           procedure BuildTree;    (* build tree from user input *)
 14   36  1  0           var FindRoot : boolean;
```

Figure 9-3. Sample Program 2A: Binary Tree Traversal

```
15   38   1  0          procedure AddNode;  (* add a node to the tree *)
16   39   2  0          begin
16   40   2  1              write(NodeCharacter : 3, NodeIndex: 3);
17   41   2  1              with ExpressionTree[NodeIndex] do begin
18   42   2  2                  Symbol:=NodeCharacter;
19   43   2  2                  read(DataFile,Left); write(Left : 3);
21   44   2  2                  read(DataFile,Right); write(Right : 3);
23   45   2  2                  readln(DataFile);
24   46   2  2                  writeln
                                  end
25   48   2  1          end; (* AddNode *)

26   50   1  0      begin
26   51   1  1          FindRoot := false;
27   52   1  1          writeln('INPUT IS:'); wrifeln;
29   53   1  1          AddNode;
30   54   1  1          repeat
30   55   1  2              read(DataFile,NodeCharacter,NodeIndex);
31   56   1  2              if NodeIndex = 1 then FindRoot := true
32   57   1  2              else AddNode
33   58   1  2          until (FindRoot) or (eof(DataFile));
35   59   1  1          writeln
                    end; (* BuildTree *)

                    (* ------------------------------------------------ *)
36   63   0  0      procedure Infix(NodeIndex : Subscr); (* write out the
                           tree in infix notation *)
37   65   1  0      begin
37   66   1  1          with ExpressionTree[NodeIndex] do
38   67   1  1              if Left <> 0 then begin
39   68   1  2                  write('(' : 1);
40   69   1  2                  Infix(Left);
41   70   1  2                  write(Symbol : 2);
42   71   1  2                  Infix(Right);
43   72   1  2                  write(')' : 1)
                                  end (* if *)
44   74   1  1              else write(Symbol : 2)
45   75   1  1      end; (* Infix *)

                    (* ------------------------------------------------ *)
46   78   0  0      procedure Prefix(NodeIndex : Subscr); (* write out the
                           tree in prefix notation *)
47   80   1  0      begin
47   81   1  1          with ExpressionTree[NodeIndex] do
48   82   1  1              if Left <> 0 then begin
49   83   1  2                  write(Symbol : 2);
50   84   1  2                  Prefix(Left);
51   85   1  2                  Prefix(Right)
                                  end   (* if *)
52   87   1  1              else write(Symbol : 2)
53   88   1  1      end; (* Prefix *)

                    (* ------------------------------------------------ *)
54   91   0  0      procedure Postfix(NodeIndex : Subscr); (* write out the
                           tree in postfix notation *)
55   93   1  0      begin
55   94   1  1          with ExpressionTree[NodeIndex] do
56   95   1  1              if Left <> 0 then begin
57   96   1  2                  Postfix(Left);
58   97   1  2                  Postfix(Right);
59   98   1  2                  write(Symbol : 2)
                                  end (* if *)
60  100   1  1              else write(Symbol : 2)
61  101   1  1      end; (* Postfix *)

                    (* ------------------------------------------------ *)
                    (* The main program reads in user data and displays the
                       tree in Infix, Prefix, and Postfix notation. *)
```

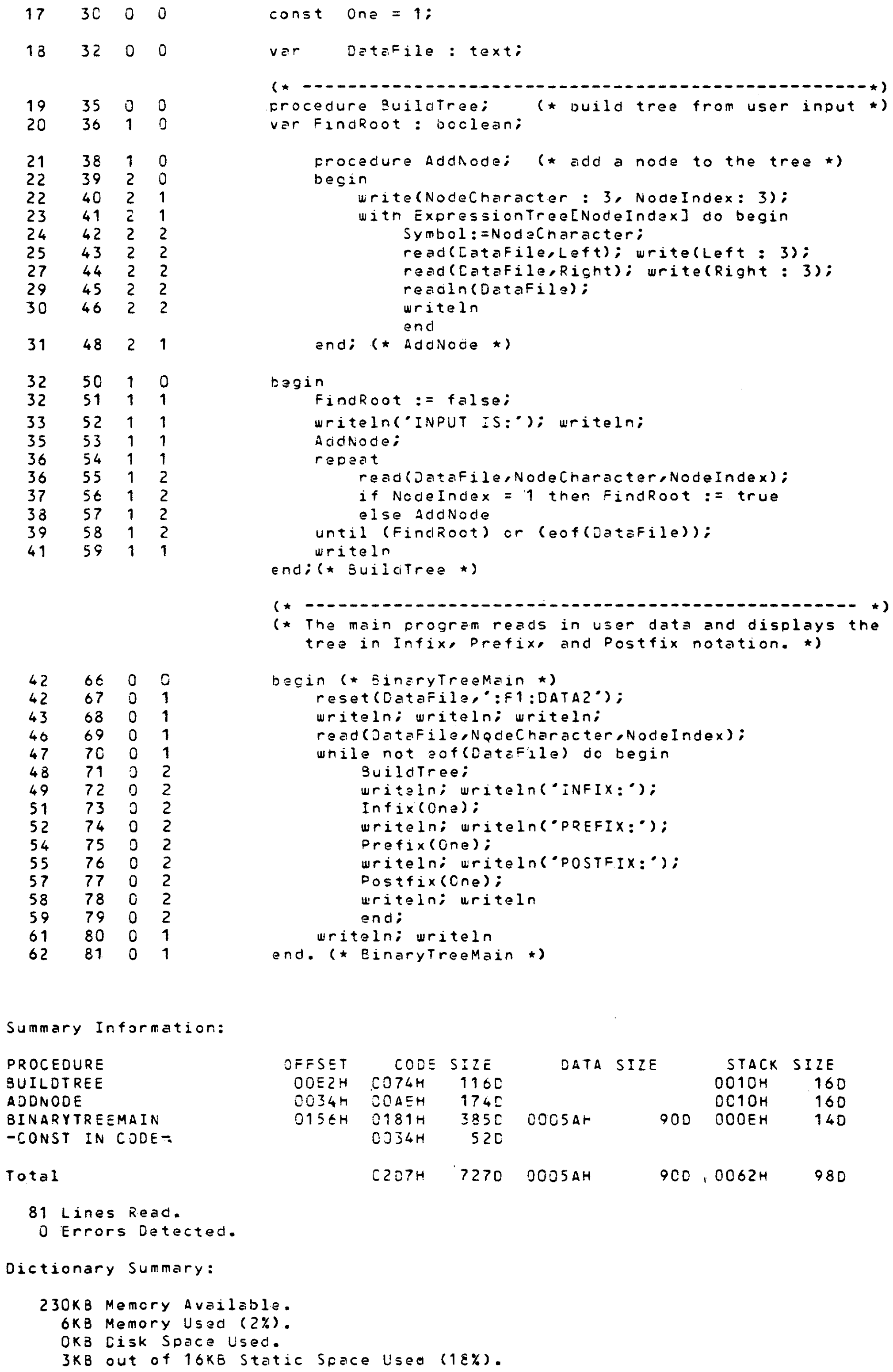
Figure 9-3. Sample Program 2A: Binary Tree Traversal (Cont'd.)

```
 62  107  0  0          begin (* TreeTraversal *)
 62  108  0  1              reset(DataFile,':F1:DATA2');
 63  109  0  1              writeln; writeln; writeln;
 66  110  0  1              read(DataFile,NodeCharacter,NodeIndex);
 67  111  0  1              while not eof(DataFile) do begin
 68  112  0  2                  BuildTree;
 69  113  0  2                  writeln; writeln('INFIX:');
 71  114  0  2                  Infix(One);
 72  115  0  2                  writeln; writeln('PREFIX:');
 74  116  0  2                  Prefix(One);
 75  117  0  2                  writeln; writeln('POSTFIX:');
 77  118  0  2                  Postfix(One);
 78  119  0  2                  writeln; writeln
 79  120  0  2                  end;
 81  121  0  1              writeln; writeln
 82  122  0  1          end. (* TreeTraversal *)


Summary Information:

PROCEDURE                 OFFSET   CODE SIZE      DATA SIZE      STACK SIZE
BUILDTREE                  00E2H   0074H   116D                  0010H    16D
ADDNODE                    0034H   00AEH   174D                  0010H    16D
INFIX                      0156H   0098H   152D                  000EH    14D
PREFIX                     01EEH   006FH   111D                  000EH    14D
POSTFIX                    025DH   006FH   111D                  000EH    14D
TREETRAVERSAL              02CCH   017BH   379D  0005AH     90D  000EH    14D
-CONST IN CODE-                    0034H    52D

Total                              0447H  1095D  0005AH     90D  008CH   140D

 122 Lines Read.
   0 Errors Detected.

Dictionary Summary:

   230KB Memory Available.
     6KB Memory Used (2%).
     0KB Disk Space Used.
     3KB out of 16KB Static Space Used (18%).
```

Figure 9-3. Sample Program 2A: Binary Tree Traversal (Cont'd.)

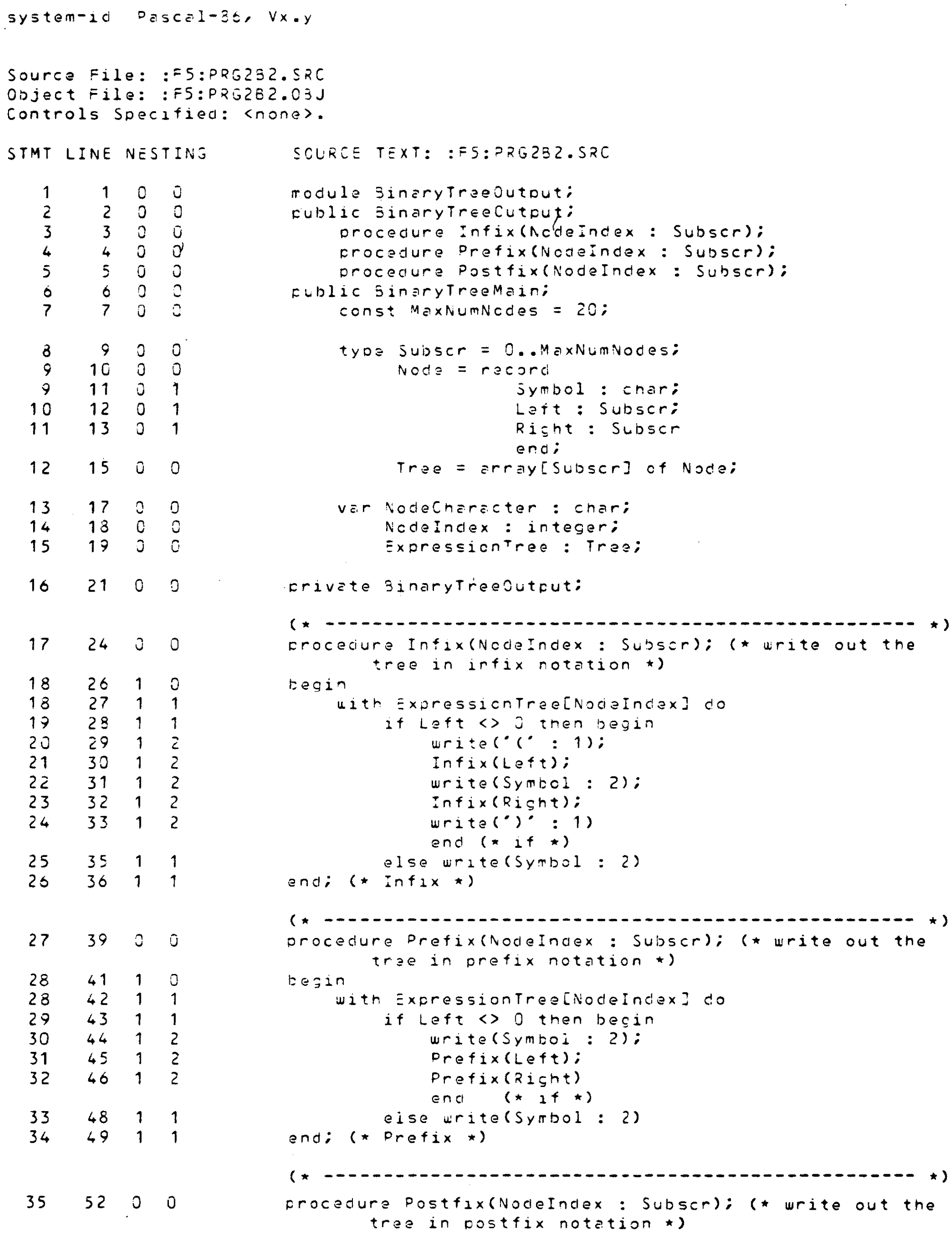
```
Source File: :F5:PRG2B1.SRC
Object File: :F5:PRG2B1.OBJ
Controls Specified: <none>.

STMT LINE NESTING       SOURCE TEXT: :F5:PRG2B1.SRC

  1    1  0  0          module BinaryTreeMain;
  2    2  0  0          public BinaryTreeOutput;
  3    3  0  0              procedure Infix(NodeIndex : Subscr);
  4    4  0  0              procedure Prefix(NodeIndex : Subscr);
  5    5  0  0              procedure Postfix(NodeIndex : Subscr);
  6    6  0  0          public BinaryTreeMain;
  7    7  0  0              const MaxNumNodes = 20;

  8    9  0  0              type Subscr = 0..MaxNumNodes;
  9   10  0  0                   Node = record
  9   11  0  1                            Symbol : char;
 10   12  0  1                            Left : Subscr;
 11   13  0  1                            Right : Subscr
                                          end;
 12   15  0  0                   Tree = array[Subscr] of Node;

 13   17  0  0              var NodeCharacter : char;
 14   18  0  0                  NodeIndex : integer;
 15   19  0  0                  ExpressionTree : Tree;

                        (* This program builds a binary tree of characters from
                        user input data and prints out the nodes of a tree in
                        infix, prefix, and postfix notation.  An input line con-
                        sists of the character, its position in the tree, and the
                        position of its two children; each item is separated from
                        the next by a blank.  *)

 16   28  0  0          program BinaryTreeMain(Input,Output);
```

Figure 9-4A. Sample Program 2B1: Binary Tree Traversal, Separately Compiled



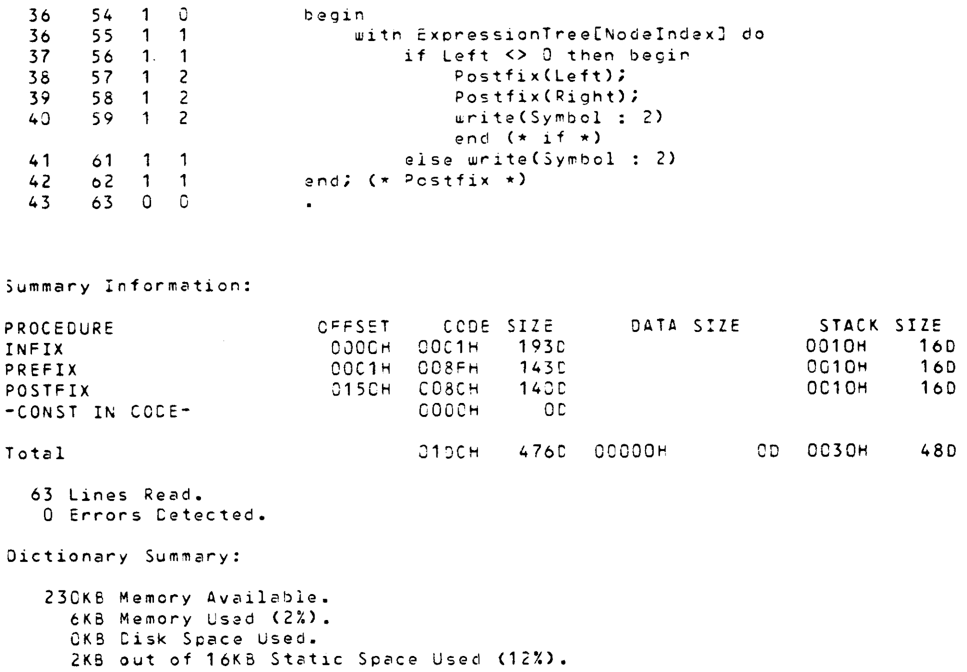
```
  17    30   0   0        const  One = 1;

  18    32   0   0        var    DataFile : text;

                          (* ---------------------------------------------------*)
  19    35   0   0        procedure BuildTree;     (* build tree from user input *)
  20    36   1   0        var FindRoot : boolean;

  21    38   1   0            procedure AddNode;  (* add a node to the tree *)
  22    39   2   0            begin
  22    40   2   1                write(NodeCharacter : 3, NodeIndex: 3);
  23    41   2   1                with ExpressionTree[NodeIndex] do begin
  24    42   2   2                    Symbol:=NodeCharacter;
  25    43   2   2                    read(DataFile,Left); write(Left : 3);
  27    44   2   2                    read(DataFile,Right); write(Right : 3);
  29    45   2   2                    readln(DataFile);
  30    46   2   2                    writeln
                                      end
  31    48   2   1            end; (* AddNode *)

  32    50   1   0        begin
  32    51   1   1            FindRoot := false;
  33    52   1   1            writeln('INPUT IS:'); writeln;
  35    53   1   1            AddNode;
  36    54   1   1            repeat
  36    55   1   2                read(DataFile,NodeCharacter,NodeIndex);
  37    56   1   2                if NodeIndex = 1 then FindRoot := true
  38    57   1   2                else AddNode
  39    58   1   2            until (FindRoot) or (eof(DataFile));
  41    59   1   1            writeln
                          end;(* BuildTree *)

                          (* -------------------------------------------------- *)
                          (* The main program reads in user data and displays the
                             tree in Infix, Prefix, and Postfix notation. *)

  42    66   0   0        begin (* BinaryTreeMain *)
  42    67   0   1            reset(DataFile,':F1:DATA2');
  43    68   0   1            writeln; writeln; writeln;
  46    69   0   1            read(DataFile,NqdeCharacter,NodeIndex);
  47    70   0   1            while not eof(DataFile) do begin
  48    71   0   2                BuildTree;
  49    72   0   2                writeln; writeln('INFIX:');
  51    73   0   2                Infix(One);
  52    74   0   2                writeln; writeln('PREFIX:');
  54    75   0   2                Prefix(One);
  55    76   0   2                writeln; writeln('POSTFIX:');
  57    77   0   2                Postfix(One);
  58    78   0   2                writeln; writeln
  59    79   0   2                end;
  61    80   0   1            writeln; writeln
  62    81   0   1        end. (* BinaryTreeMain *)


Summary Information:

PROCEDURE                OFFSET    CODE SIZE       DATA SIZE       STACK SIZE
BUILDTREE                00E2H    0074H    116D                    0010H    16D
ADDNODE                  0034H    00AEH    174D                    0010H    16D
BINARYTREEMAIN           0156H    0181H    385D   0005AH     90D   000EH    14D
-CONST IN CODE-                   0034H     52D

Total                             02D7H    727D   0005AH     90D   0062H    98D

  81 Lines Read.
   0 Errors Detected.

Dictionary Summary:

   230KB Memory Available.
     6KB Memory Used (2%).
     0KB Disk Space Used.
     3KB out of 16KB Static Space Used (18%).
```

Figure 9-4A. Sample Program 2B1: Binary Tree Traversal, Separately Compiled (Cont'd.)



```
system-id  Pascal-86, Vx.y

Source File: :F5:PRG2B2.SRC
Object File: :F5:PRG2B2.OBJ
Controls Specified: <none>.

STMT LINE NESTING        SOURCE TEXT: :F5:PRG2B2.SRC

   1     1  0  0         module BinaryTreeOutput;
   2     2  0  0         public BinaryTreeOutput;
   3     3  0  0             procedure Infix(NodeIndex : Subscr);
   4     4  0  0             procedure Prefix(NodeIndex : Subscr);
   5     5  0  0             procedure Postfix(NodeIndex : Subscr);
   6     6  0  0         public BinaryTreeMain;
   7     7  0  0             const MaxNumNodes = 20;

   8     9  0  0             type Subscr = 0..MaxNumNodes;
   9    10  0  0                  Node = record
   9    11  0  1                            Symbol : char;
  10    12  0  1                            Left : Subscr;
  11    13  0  1                            Right : Subscr
                                            end;
  12    15  0  0                  Tree = array[Subscr] of Node;

  13    17  0  0             var NodeCharacter : char;
  14    18  0  0                 NodeIndex : integer;
  15    19  0  0                 ExpressionTree : Tree;

  16    21  0  0         private BinaryTreeOutput;

                         (* ---------------------------------------------------- *)
  17    24  0  0         procedure Infix(NodeIndex : Subscr); (* write out the
                                 tree in infix notation *)
  18    26  1  0         begin
  18    27  1  1             with ExpressionTree[NodeIndex] do
  19    28  1  1                 if Left <> 0 then begin
  20    29  1  2                     write('(' : 1);
  21    30  1  2                     Infix(Left);
  22    31  1  2                     write(Symbol : 2);
  23    32  1  2                     Infix(Right);
  24    33  1  2                     write(')' : 1)
                                     end (* if *)
  25    35  1  1                 else write(Symbol : 2)
  26    36  1  1         end; (* Infix *)

                         (* ---------------------------------------------------- *)
  27    39  0  0         procedure Prefix(NodeIndex : Subscr); (* write out the
                                 tree in prefix notation *)
  28    41  1  0         begin
  28    42  1  1             with ExpressionTree[NodeIndex] do
  29    43  1  1                 if Left <> 0 then begin
  30    44  1  2                     write(Symbol : 2);
  31    45  1  2                     Prefix(Left);
  32    46  1  2                     Prefix(Right)
                                     end   (* if *)
  33    48  1  1                 else write(Symbol : 2)
  34    49  1  1         end; (* Prefix *)

                         (* ---------------------------------------------------- *)
  35    52  0  0         procedure Postfix(NodeIndex : Subscr); (* write out the
                                 tree in postfix notation *)
```

Figure 9-4B. Sample Program 2B2: Binary Tree Traversal, Separately Compiled



```
36   54  1  0         begin
36   55  1  1             with ExpressionTree[NodeIndex] do
37   56  1. 1                 if Left <> 0 then begin
38   57  1  2                     Postfix(Left);
39   58  1  2                     Postfix(Right);
40   59  1  2                     write(Symbol : 2)
                                  end (* if *)
41   61  1  1                 else write(Symbol : 2)
42   62  1  1         end; (* Postfix *)
43   63  0  0         .


Summary Information:

PROCEDURE                   OFFSET   CODE SIZE       DATA SIZE       STACK SIZE
INFIX                        000CH   00C1H   193D                    0010H    16D
PREFIX                       00C1H   008FH   143D                    0010H    16D
POSTFIX                      015CH   008CH   140D                    0010H    16D
-CONST IN CODE-                      0000H     0D

Total                                01DCH   476D  00000H         0D 0030H    48D

  63 Lines Read.
   0 Errors Detected.

Dictionary Summary:

   230KB Memory Available.
     6KB Memory Used (2%).
     0KB Disk Space Used.
     2KB out of 16KB Static Space Used (12%).
```

Figure 9-4B. Sample Program 2B2: Binary Tree Traversal, Separately Compiled (Cont'd.)

9.3 Sample Program 3: Quadratic Roots

The program of figure 9-5 computes the non-complex roots of a series of quadratic equations of the form

$y=ax^2+bx+c$

The program takes the quadratic coefficients a, b, and c from the file **data**, and outputs the roots to the file **quad**. When the roots to an equation have an imaginary part, the program does not compute the roots, but writes a message to the file RESULT on device :F1:.

This program illustrates the use of REAL and TEMPREAL numbers, a non-text file, and file input and output via GET and PUT.

Note that this program, like any other that performs input of numeric values using GET rather than READ, cannot directly handle console input. This is because console I/O is in the form of text files (i.e., files of CHAR values organized into lines), and GET does no type conversions. The program assumes that the file **data** is already in the correct (REAL) format.

This program must be linked to CEL87.LIB, the run-time support libraries (P86RN0.LIB, P86RN1.LIB, P86RN2.LIB, P86RN3.LIB), the appropriate 8087 libraries (either 8087.LIB, or E8087.LIB and E8087), and any interface libraries required by your operating system.

```
system-id  Pascal-86, Vx.y

Source File: :F5:PROG3.SRC
Object File: :F5:PROG3.OBJ
Controls Specified: <none>.

STMT LINE NESTING        SOURCE TEXT: :F5:PROG3.SRC

                         (*  This program computes the non-complex roots of a quadratic equation
                         of the form a*x**2 + b*x + c.  It performs I/O to a binary file, but
                         displays the input coefficients and outputs roots in a readable form
                         on the console.  If the roots are imaginary, the program prints an
                         appropriate message.

                            Variables:
                              ZeroConst -    real constant used to check for imaginary roots
                              InputCoef -    real input file which holds the input coefficients
                              QuadResult -   real output file which holds the quadratic results
                              AVar,BVar,CVar - real coefficients
                              TempVar -      real value for temporary storage     *)

   1   15  0  0          program Quadratic(Output);

   2   17  0  0          const ZeroConst = 0.0;

   3   19  0  0          var InputCoef,QuadResult : file of real;
   4   20  0  0              AVar,BVar,CVar : real;
   5   21  0  0              TempVar : TEMPREAL;

   6   23  0  0          begin

   6   25  0  1              reset(InputCoef,':F1:DATA3');    (* position input file at beginning *)
   7   26  0  1              rewrite(QuadResult,':F1:RESULT');  (* initialize output file *)

   8   28  0  1              while not EOF(InputCoef) do begin

   9   30  0  2                  AVar := InputCoef^;   get(InputCoef);          (* get *)
  11   31  0  2                  BVar := InputCoef^;   get(InputCoef);      (* coefficients *)
  13   32  0  2                  CVar := InputCoef^;   get(InputCoef);    (* from input file *)
  15   33  0  2                  write('Input coefficients are: ',AVar,BVar,CVar);  (* echo inputs *)
  16   34  0  2                  writeln; write('Roots are: ');
  18   35  0  2                  TempVar := (BVar * BVar) - (4 * AVar * CVar); (* compute discriminant *)
  19   36  0  2                  if TempVar >= ZeroConst then begin
  20   37  0  3                      TempVar := sqrt(TempVar);
  21   38  0  3                      QuadResult^ := (-BVar + TempVar) / (2 * AVar);
  22   39  0  3                      write(QuadResult^);  put(QuadResult);
  24   40  0  3                      QuadResult^ := -(BVar + TempVar) / (2 * AVar);
  25   41  0  3                      writeln(QuadResult^);   put(QuadResult)
  26   42  0  3                      end    (* of if *)
  27   43  0  2                  else writeln(' imaginary'); (* print to default output file *)

  29   45  0  2              end;  (* of while *)

  31   47  0  1              writeln
                         end.(* Quadratic *)


Summary Information:

PROCEDURE                      OFFSET    CODE SIZE       DATA SIZE       STACK SIZE
QUADRATIC                       0051H  026CH    608D  00032H        50D  000EH     14D
-CONST IN CODE-                        0051H     81D

Total                                  0281H    689D  00032H        50D  0042H     66D

   49 Lines Read.
    0 Errors Detected.

Dictionary Summary:

   230KB Memory Available.
     6KB Memory Used (2%).
     0KB Disk Space Used.
     3KB out of 16KB Static Space Used (18%).
```

Figure 9-5. Sample Program 3: Quadratic Roots

9.4 Sample Program 4: Text Editor

The program of figure 9-6 is a simple text editor. It reads a preconnected file of characters and echoes the input to the OUTPUT file, recognizing the following control characters and performing the indicated actions:

( ) Ignore all enclosed characters.
* Start new output line.
/ Exchange preceding and following characters.
+ Delete preceding character.

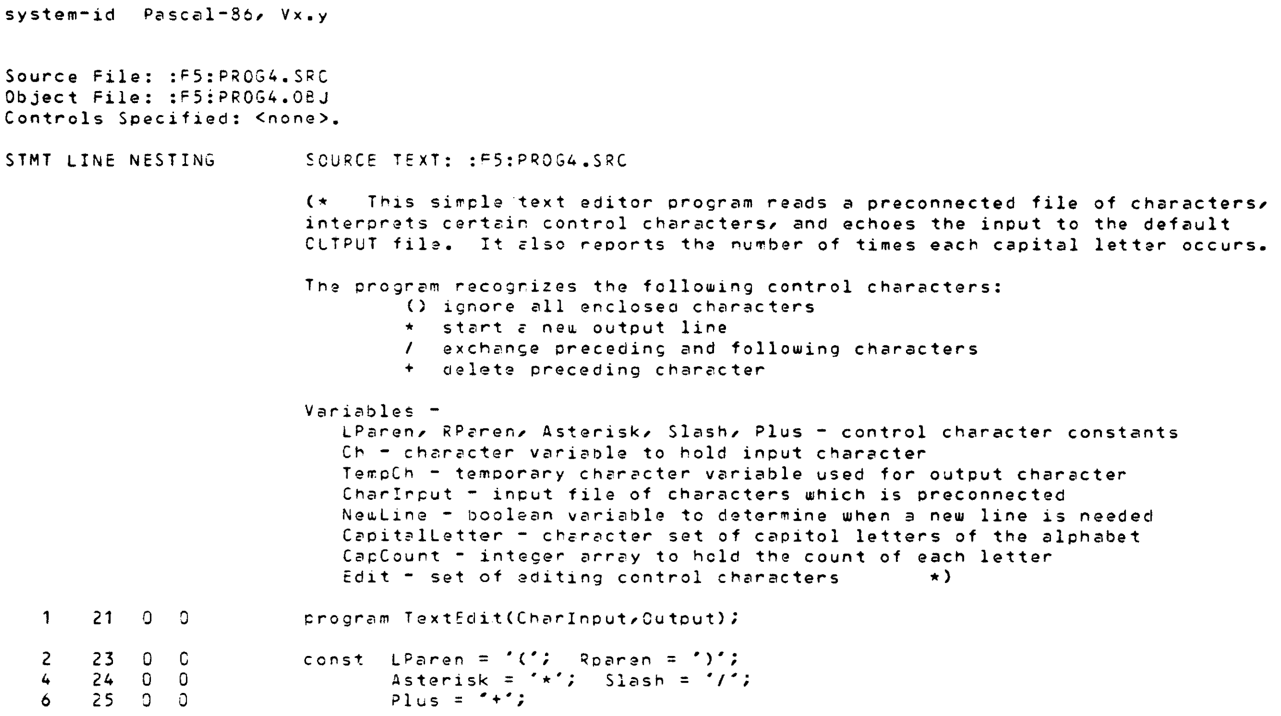
The control characters themselves do not appear in the output. In addition, the program counts the number of occurrences of each capital letter in the input and prints out the result following the text output.

Note that the input file is a text file; therefore, carriage returns and line feeds are read as blanks. The output file (the standard file OUTPUT) is, of course, a text file.

This program illustrates the use of arrays, sets, Boolean variables and preconnection, as well as file input and output. Execute this program with the invocation line:

```
PROG4(CHARINPUT=DATA4)<cr>
```

This program must be linked to the run-time support libraries (P86RN0.LIB, P86RN1.LIB, P86RN2.LIB, P86RN3.LIB), 87NULL.LIB, and any interface libraries required by your operating system.



```
system-id   Pascal-86, Vx.y

Source File: :F5:PROG4.SRC
Object File: :F5:PROG4.OBJ
Controls Specified: <none>.

STMT LINE NESTING            SOURCE TEXT: :F5:PROG4.SRC

                             (*  This simple text editor program reads a preconnected file of characters,
                             interprets certain control characters, and echoes the input to the default
                             OUTPUT file.  It also reports the number of times each capital letter occurs.

                             The program recognizes the following control characters:
                                   () ignore all enclosed characters
                                   *  start a new output line
                                   /  exchange preceding and following characters
                                   +  delete preceding character

                             Variables -
                                LParen, RParen, Asterisk, Slash, Plus - control character constants
                                Ch - character variable to hold input character
                                TempCh - temporary character variable used for output character
                                CharInput - input file of characters which is preconnected
                                NewLine - boolean variable to determine when a new line is needed
                                CapitalLetter - character set of capitol letters of the alphabet
                                CapCount - integer array to hold the count of each letter
                                Edit - set of editing control characters      *)

   1    21   0   0           program TextEdit(CharInput,Output);

   2    23   0   0           const  LParen = '(';  Rparen = ')';
   4    24   0   0                  Asterisk = '*';  Slash = '/';
   6    25   0   0                  Plus = '+';
```

Figure 9-6. Sample Program 4: Text Editor

```
  7   27  0  0          var    Ch,TempCh : char;
  8   28  0  0                 CharInput : text;
  9   29  0  0                 NewLine : boolean;
 10   30  0  0                 CapitalLetter : set of 'A'..'Z';
 11   31  0  0                 CapCount : array['A'..'Z'] of integer;
 12   32  0  0                 Edit : set of Lparen..Slash;

 13   34  0  0          begin
 13   35  0  1              CapitalLetter := ['A'..'Z'];              (* perform initialization *)
 14   36  0  1              Edit := ['(','+','*','/'];
 15   37  0  1              for Ch := 'A' to 'Z' do CapCount[Ch] := 0;
 17   38  0  1              reset(CharInput);
 18   39  0  1              TempCh := '*';

 19   41  0  1              while not eof(CharInput) do begin
 20   42  0  2                  read(CharInput,Ch);            (* input a character *)
 21   43  0  2                  if [Ch] * CapitalLetter <> [] then CapCount[Ch] := CapCount[Ch] + 1;
 23   44  0  2                  if [Ch] * Edit <> [] then
 24   45  0  2                    case Ch of                       (* action for control characters *)
 25   46  0  3                        LParen : begin                 (* ignore embedded text *)
 25   47  0  4                            repeat read(CharInput,Ch) until Ch = RParen;
 27   48  0  4                            read(CharInput,Ch)
                                          end;
 29   50  0  3                        Asterisk : NewLine := true;     (* flag a new line *)
 30   51  0  3                        Plus : read(CharInput,TempCh); (* delete preceding character *)
 31   52  0  3                        Slash : begin                   (* perform character exchange *)
 31   53  0  4                            Ch := TempCh;
 32   54  0  4                            read(CharInput,TempCh);
 33   55  0  4                            end
 34   56  0  3                    end;(* case *)
 36   57  0  2                  if [TempCh] * Edit = [] then write(TempCh);
 38   58  0  2                  if NewLine then begin
 39   59  0  3                      writeln;
 40   60  0  3                      NewLine := false
                                    end; (* if *)
 42   62  0  2                  TempCh := Ch                  (* assign output character for next loop *)
                                end; (* while *)

 44   65  0  1              writeln(TempCh); writeln;          (* write out last character *)
 46   66  0  1              for Ch := 'A' to 'Z' do write(Ch : 3,CapCount[Ch] : 2);
 48   67  0  1              writeln
                        end. (* TextEdit *)


Summary Information:

PROCEDURE                     OFFSET    CODE SIZE       DATA SIZE       STACK SIZE
TEXTEDIT                       0011H   02BFH    703D  00054H       84D  000EH     14D
-CONST IN CODE-                        0011H     17D

Total                                  02D0H    720D  00054H       84D  0042H     66D

  68 Lines Read.
   0 Errors Detected.

Dictionary Summary:

   230KB Memory Available.
     6KB Memory Used (2%).
     0KB Disk Space Used.
     4KB out of 16KB Static Space Used (25%).
```

Figure 9-6. Sample Program 4: Text Editor (Cont'd.)

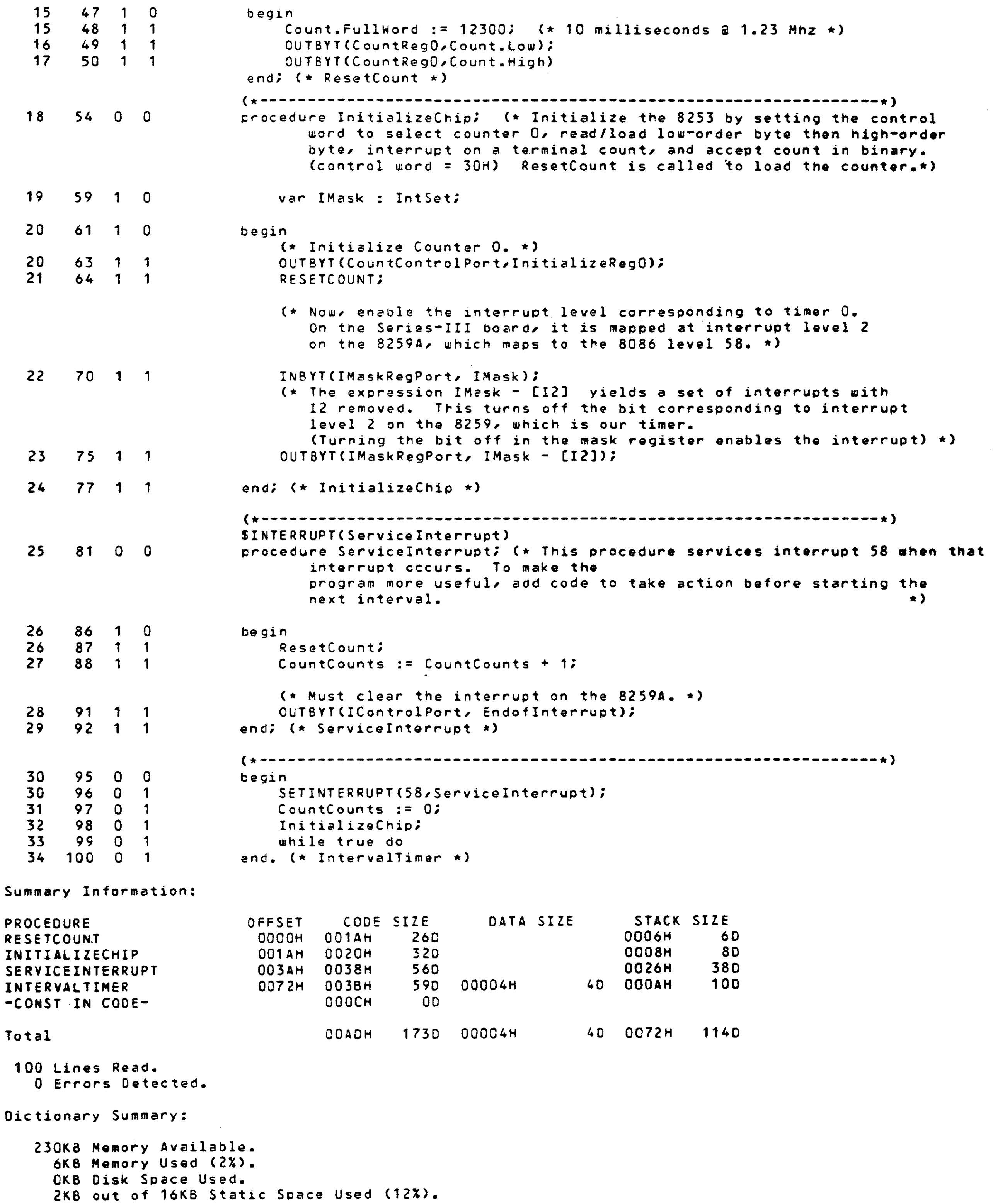
9.5 Sample Program 5: Interrupt Processing

The program of figure 9-7, designed to run on an iSBC 86/12A system rather than on a development system, initializes the 8253 interval timer on the iSBC 86/12A board to interrupt the host processor every 10 milliseconds. This program illustrates the use of Pascal-86 predefined procedures for port input/output (INBYT and OUTBYT) and interrupt control (SETINTERRUPT and CAUSEINTERRUPT). It also shows how to bypass Pascal's type checking using variant records, and how to manipulate bits using sets.

Since this program is designed to run in a bare machine (non-operating system) environment, it need only be linked with the run-time support libraries RTNULL,LIB,P86RN1.LIB (in that order) and 87NULL.LIB.

```
 system-id  Pascal-86, Vx.y


Source File: :F5:PROG5.SRC
Object File: :F5:PROG5.OBJ
Controls Specified: <none>.

STMT LINE NESTING         SOURCE TEXT: :F5:PROG5.SRC

                          (*  This program uses an 8253 interval timer on an iSBC 86/12A
                          board to interrupt the host processor every 10 milliseconds.
                          It assumes the 86/12A board is set up as a standard Series-III board
                          is set up, with:
                             - counter 0 of the on-board 8253 free, and its interrupt strapped
                               to level 2 of the on-board 8259A.
                             - The on-board 8259A initialized as on the Series-III, with its
                               8 interrupts mapped in from 56 to 63.
                          *)

   1   11  0  0           program IntervalTimer;

                            (*  8253 Port address definitions. *)
   2   14  0  0            const CountControlPort = 006H;
   3   15  0  0                  CountReg0        = 000H;
   4   16  0  0                  InitializeReg0   = 030H;

                            (* 8259a Port address and control word definitions. *)
   5   19  0  0                  IMaskRegPort     = 0C2H;
   6   20  0  0                  IControlPort     = 0C0H;
   7   21  0  0                  EndofInterrupt   = 020H;

                            (* Define a set of interrupts to use to mask the 8259A interrupt mask *)
                            (* The 8259A assigns interrupt levels from right to left in its control
                               register, the same order that Pascal-86 assigns bits to set elements.
                             *)

   8   28  0  0              type  IntLevels = (I0, I1, I2, I3, I4, I5, I6, I7);
   9   29  0  0                    IntSet    = Set of IntLevels;

  10   31  0  0              var   CountCounts : LONGINT;  (* Extend timer here. Holds count of
                                                              10 msec. intervals recieved. *)
                          (*---------------------------------------------------------------*)
  11   34  0  0           procedure ResetCount;
                                (* This procedure loads the counter value with 12300
                                so that the counter will count down for exactly 10
                                milliseconds. ( The iSBC 86/12A clock rate is 1.23MHz.)  *)


                          (* Use a variant record to map two bytes "on top of" a word.
                          Note that the low byte precedes the high byte. *)
  12   42  1  0           var   Count : RECORD CASE BOOLEAN OF
  13   43  1  1                          true:( FullWord: WORD);
  14   44  1  1                         false:( Low, High: 0..255)
                                        END;
```

Figure 9-7. Sample Program 5: Interrupt Processing



```
  15    47  1  0           begin
  15    48  1  1               Count.FullWord := 12300;  (* 10 milliseconds @ 1.23 Mhz *)
  16    49  1  1               OUTBYT(CountReg0,Count.Low);
  17    50  1  1               OUTBYT(CountReg0,Count.High)
                           end; (* ResetCount *)
                          (*---------------------------------------------------------------*)
 18    54  0  0           procedure InitializeChip;  (* Initialize the 8253 by setting the control
                                 word to select counter 0, read/load low-order byte then high-order
                                 byte, interrupt on a terminal count, and accept count in binary.
                                 (control word = 30H)  ResetCount is called to load the counter.*)

 19    59  1  0               var IMask : IntSet;

 20    61  1  0           begin
                              (* Initialize Counter 0. *)
 20    63  1  1               OUTBYT(CountControlPort,InitializeReg0);
 21    64  1  1               RESETCOUNT;

                              (* Now, enable the interrupt level corresponding to timer 0.
                                 On the Series-III board, it is mapped at interrupt level 2
                                 on the 8259A, which maps to the 8086 level 58. *)

 22    70  1  1               INBYT(IMaskRegPort, IMask);
                              (* The expression IMask - [I2]  yields a set of interrupts with
                                 I2 removed.  This turns off the bit corresponding to interrupt
                                 level 2 on the 8259, which is our timer.
                                 (Turning the bit off in the mask register enables the interrupt) *)
 23    75  1  1               OUTBYT(IMaskRegPort, IMask - [I2]);

 24    77  1  1           end; (* InitializeChip *)

                          (*---------------------------------------------------------------*)
                          $INTERRUPT(ServiceInterrupt)
 25    81  0  0           procedure ServiceInterrupt; (* This procedure services interrupt 58 when that
                                 interrupt occurs.  To make the
                                 program more useful, add code to take action before starting the
                                 next interval.                                              *)

 26    86  1  0           begin
 26    87  1  1               ResetCount;
 27    88  1  1               CountCounts := CountCounts + 1;

                              (* Must clear the interrupt on the 8259A. *)
 28    91  1  1               OUTBYT(IControlPort, EndofInterrupt);
 29    92  1  1           end; (* ServiceInterrupt *)

                          (*---------------------------------------------------------------*)
 30    95  0  0           begin
 30    96  0  1               SETINTERRUPT(58,ServiceInterrupt);
 31    97  0  1               CountCounts := 0;
 32    98  0  1               InitializeChip;
 33    99  0  1               while true do
 34   100  0  1           end. (* IntervalTimer *)

Summary Information:

PROCEDURE                 OFFSET    CODE SIZE      DATA SIZE      STACK SIZE
RESETCOUNT                 0000H   001AH    26D                   0006H      6D
INITIALIZECHIP             001AH   0020H    32D                   0008H      8D
SERVICEINTERRUPT           003AH   0038H    56D                   0026H     38D
INTERVALTIMER              0072H   003BH    59D  00004H       4D  000AH     10D
-CONST IN CODE-                    000CH     0D

Total                              00ADH   173D  00004H       4D  0072H    114D

 100 Lines Read.
   0 Errors Detected.

Dictionary Summary:

   230KB Memory Available.
     6KB Memory Used (2%).
     0KB Disk Space Used.
     2KB out of 16KB Static Space Used (12%).
```

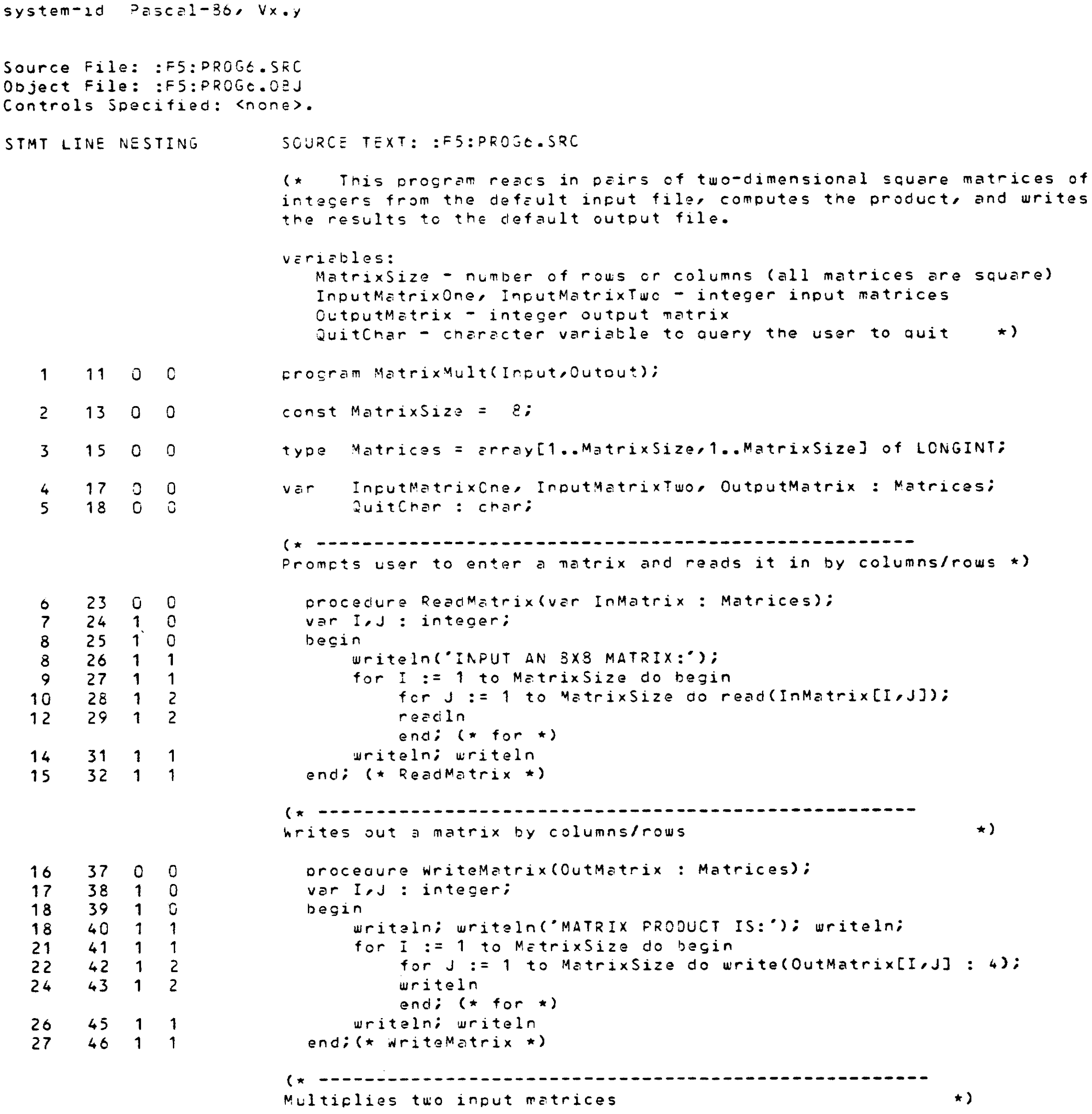
Figure 9-7. Sample Program 5: Interrupt Processing (Cont'd.)

9.6 Sample Program 6: Matrix Multiplication

The program of figure 9-8 reads in pairs of eight-by-eight two-dimensional matrices of integers from the file INPUT, computes the product, and writes the output to the file OUTPUT. It illustrates the use of multi-dimensional arrays, value and variable parameters, array arguments, and text file input and output.

The algorithm used in this program was chosen for its simplicity. It may not be mathematically optimal.

This program must be linked to the run-time support libraries (P86RN0.LIB, P86RN1.LIB, P86RN2.LIB, P86RN3.LIB), 87NULL.LIB, and any interface libraries required by your operating system.



```
system-id  Pascal-86, Vx.y


Source File: :F5:PROG6.SRC
Object File: :F5:PROG6.OBJ
Controls Specified: <none>.

STMT LINE NESTING          SOURCE TEXT: :F5:PROG6.SRC

                           (*  This program reads in pairs of two-dimensional square matrices of
                           integers from the default input file, computes the product, and writes
                           the results to the default output file.

                           variables:
                              MatrixSize - number of rows or columns (all matrices are square)
                              InputMatrixOne, InputMatrixTwo - integer input matrices
                              OutputMatrix - integer output matrix
                              QuitChar - character variable to query the user to quit    *)

  1   11  0  0             program MatrixMult(Input,Output);

  2   13  0  0             const MatrixSize =  8;

  3   15  0  0             type  Matrices = array[1..MatrixSize,1..MatrixSize] of LONGINT;

  4   17  0  0             var   InputMatrixOne, InputMatrixTwo, OutputMatrix : Matrices;
  5   18  0  0                   QuitChar : char;

                           (* ---------------------------------------------------
                           Prompts user to enter a matrix and reads it in by columns/rows *)

  6   23  0  0               procedure ReadMatrix(var InMatrix : Matrices);
  7   24  1  0               var I,J : integer;
  8   25  1  0               begin
  8   26  1  1                  writeln('INPUT AN 8X8 MATRIX:');
  9   27  1  1                  for I := 1 to MatrixSize do begin
 10   28  1  2                      for J := 1 to MatrixSize do read(InMatrix[I,J]);
 12   29  1  2                      readln
                                    end; (* for *)
 14   31  1  1                  writeln; writeln
 15   32  1  1               end; (* ReadMatrix *)

                           (* ---------------------------------------------------
                           Writes out a matrix by columns/rows                          *)

 16   37  0  0               procedure WriteMatrix(OutMatrix : Matrices);
 17   38  1  0               var I,J : integer;
 18   39  1  0               begin
 18   40  1  1                  writeln; writeln('MATRIX PRODUCT IS:'); writeln;
 21   41  1  1                  for I := 1 to MatrixSize do begin
 22   42  1  2                      for J := 1 to MatrixSize do write(OutMatrix[I,J] : 4);
 24   43  1  2                      writeln
                                    end; (* for *)
 26   45  1  1                  writeln; writeln
 27   46  1  1               end;(* WriteMatrix *)

                           (* ---------------------------------------------------
                           Multiplies two input matrices                                *)
```

Figure 9-8. Sample Program 6: Matrix Multiplication

```
28    51   0   0          procedure Product(var ProdMatrix : Matrices;
28    52   1   0                     OneMatrix,TwoMatrix : Matrices);
29    53   1   0          var I, J, K : integer;
30    54   1   0            Result  : LONGINT;
31    55   1   0          begin
31    56   1   1            for I := 1 to MatrixSize do
32    57   1   1              for J := 1 to MatrixSize do begin
33    58   1   2                Result := 0;
34    59   1   2                for K := 1 to MatrixSize do
35    60   1   2                  Result := Result + OneMatrix[I,K] * TwoMatrix[K,J];
36    61   1   2                ProdMatrix[I,J] := Result
                                  end (* for *)
37    63   1   1          end; (* Product *)

                         (* ---------------------------------------------------*)
38    66   0   0         begin (* MultMatrix *)
38    67   0   1           repeat
38    68   0   2             ReadMatrix(InputMatrixOne);    (* input first matrix *)
39    69   0   2             ReadMatrix(InputMatrixTwo);    (* input second matrix *)
40    70   0   2             Product(OutputMatrix,InputMatrixOne,InputMatrixTwo); (* multiply them *)
41    71   0   2             WriteMatrix(OutputMatrix);     (* output the resulting matrix *)
42    72   0   2             write('ANOTHER MATRIX?  ');    (* query for another matrix *)
43    73   0   2             read(QuitChar); writeln
44    74   0   2           until not (QuitChar in ['Y','y'])
45    75   0   2         end. (* MultMatrix *)


Summary Information:

PROCEDURE                     OFFSET    CODE SIZE        DATA SIZE        STACK SIZE
READMATRIX                     0044H   0085H   133D                      001AH    26D
WRITEMATRIX                    00C9H   0099H   153D                      001AH    26D
PRODUCT                        0162H   00A7H   167D                      001EH    30D
MATRIXMULT                     02C9H   0110H   272D  00311H      785D    0206H   518D
-CONST IN CODE-                        0044H    68D

Total                                  0319H   793D  00311H      785D    028CH   652D

  75 Lines Read.
   0 Errors Detected.

Dictionary Summary:

   230KB Memory Available.
     6KB Memory Used (2%).
     0KB Disk Space Used.
     4KB out of 16KB Static Space Used (25%).
```

Figure 9-8. Sample Program 6: Matrix Multiplication (Cont'd.)

9.7 Sample Program 7: Maze Game

The program of figure 9-9 finds a path through a maze and writes out the solution. The maze is a 7 by 7 square consisting of passageways (represented by dots) and walls (represented by W's). For example:

```
...W.W.
.WW.WWW
.W..W..
...W...
..W..W.
..W.W..
....W..
```

The starting point is always the top left corner, and the exit is the bottom right corner. Movement through the maze can be vertical or horizontal, but not diagonal. If a path through the maze exists, the program prints out a square matrix in which up-arrow ↑ symbols represent the path, X's represent points visited that are not part of the path, dots represent passageway points that were not visited, and W's represent walls. For example, the output corresponding to the sample input above is:

```
↑ X X W . W .
↑ W W X W W W
↑ W X X W . .
↑ ↑ X W ↑ ↑ ↑
. ↑ W ↑ ↑ W ↑
. ↑ W ↑ W . ↑
. ↑ ↑ ↑ W . ↑
```

If there is no possible path through the maze, the program writes out a message in place of the solution. You must input the maze configuration without any blanks. Blanks are added to the output for readability.

The procedure **findpath**, which does the work of finding a way through the maze, is directly recursive—that is, it calls itself. This program must be linked to the run-time support libraries (P86RN0.LIB, P86RN1.LIB, P86RN2.LIB, P86RN3.LIB), 87NULL.LIB, and any interface libraries required by your operating system.

```
system-id   Pascal-86, Vx.y

Source File: :F5:PROG7.SRC
Object File: :F5:PROG7.OBJ
Controls Specified: <none>.

STMT LINE NESTING           SOURCE TEXT: :F5:PROG7.SRC

                            (*  This maze game program first prompts the user to enter a seven-by-seven
                            maze, where a '.' indicates a path and a 'W' represents a wall.  The program
                            then determines whether or not there is a way out by beginning in the upper
                            left-hand corner and trying to exit at the lower right-hand corner.  The
                            program marks with 'X's the trail which actually leads out, if any.  If there
                            is no way out of the maze, the program displays an appropriate message.  Other-
                            wise it displays the final maze and prompts the user to enter another maze. *)

   1     9  0  0           program AMazeGame(Input,Output);

   2    11  0  0           const  Top = 1;           (* constant to mark the smallest row number *)
   3    12  0  0                  Bottom = 7;        (* constant to mark the largest row number *)
   4    13  0  0                  Left = 1;          (* constant to mark the smallest column number *)
   5    14  0  0                  Right = 7;         (* constant to mark the largest column number *)
   6    15  0  0                  Path = '.';        (* constant to mark a path input by user *)
   7    16  0  0                  Trail = '^';       (* constant to mark the trail the program took *)
   8    17  0  0                  Mark = 'x';        (* constant to mark the path which was travelled *)
   9    18  0  0                  Size = 7;          (* number of columns or rows in the maze *)

  10    20  0  0           var    Maze : array[1..Size,1..Size] of char;   (* array to hold maze *)
  11    21  0  0                  WayOut : boolean;                 (* flag to indicate a way out *)
  12    22  0  0                  InputChar : char;                 (* prompt user to play again *)

                           (*---------------------------------------------*)
  13    25  0  0             procedure ReadMaze;        (* procedure to input the user's maze *)
  14    26  1  0             var I, J  : 1..Size;
  15    27  1  0             begin
  15    28  1  1                 writeln('INPUT A MAZE:'); writeln;
  17    29  1  1                 for I := 1 to Size do begin
  18    30  1  2                     for J := 1 to Size do read(Maze[I,J]);
  20    31  1  2                     readln
                                     end (* for *)
  21    33  1  1             end;(* ReadMaze *)
```

Figure 9-9. Sample Program 7: Maze Game

```
                  (*---------------------------------------------*)
22   36   0   0     procedure WriteMaze;         (* procedure to output the maze *)
23   37   1   0     var I, J : 1..Size;
24   38   1   0     begin
24   39   1   1         writeln('PATH THROUGH MAZE IS:'); writeln;
26   40   1   1         for I:= 1 to Size do begin
27   41   1   2             for J := 1 to Size do write(Maze[I,J] : 2);
29   42   1   2             writeln
                            end (* for *)
30   44   1   1     end; (* WriteMaze *)

                  (*---------------------------------------------*)
                  (* Recursive procedure to find a path through the maze.  This procedure
                  continues to call itself until either the way out has been found or it has
                  determined there is no way out.  The algorithm begins at the upper right-hand
                  corner of the maze and tries to move to the lower left-hand corner, marking
                  the path it has traveled with an X.  The algorithm first tries to move
                  right, then down, then left, then up.  If there is a way out, the procedure
                  marks the trail with up-arrow symbols as the recursion levels decrease.

                        Row - parameter to indicate row number
                        Column - parameter to indicate column number     *)

31   58   0   0     procedure FindPath(Row, Column : integer);
32   59   1   0     begin
32   60   1   1         if (Row = Bottom) and (Column = Right)
                            then WayOut := true
33   62   1   1         else begin
34   63   1   2             Maze[Row,Column] := Mark;
35   64   1   2             if Column <> Right then
36   65   1   2                 if Maze[Row,Column + 1] = Path then
37   66   1   2                     FindPath(Row,Column + 1);         (* move right if possible *)
38   67   1   2             if (not WayOut) and (Row <> Bottom) then
39   68   1   2                 if Maze[Row + 1,Column] = Path then
40   69   1   2                     FindPath(Row + 1,Column);         (* move down if possible *)
41   70   1   2             if (not WayOut) and (Column <> Left) then
42   71   1   2                 if Maze[Row,Column - 1] = Path then
43   72   1   2                     FindPath(Row,Column - 1);         (* move left if possible *)
44   73   1   2             if (not WayOut) and (Row <> Top) then
45   74   1   2                 if Maze[Row - 1,Column] = Path then
46   75   1   2                     FindPath(Row - 1,Column)          (* move up if possible *)
                            end; (* else *)
48   77   1   1         if WayOut then Maze[Row,Column] := Trail
49   78   1   1     end; (* FindPath *)

                  (*---------------------------------------------*)
                  (* The main program inputs a maze, initially assumes there is no way out, then
                  calls the recursive procedure FindPath to find a way out if there is one.  If
                  there is no way out, the program prints a message; otherwise, it prints the
                  final maze. *)

50   86   0   0   begin  (* AMazeGame *)
50   87   0   1       repeat
50   88   0  `2           ReadMaze;
51   89   0   2           WayOut := false;
52   90   0   2           FindPath(Top,Left);
53   91   0   2           if WayOut then WriteMaze
54   92   0   2           else begin      (* if no way out, print message *)
55   93   0   3               writeln; writeln('NO PATH THROUGH MAZE FOUND.');
57   94   0   3               writeln
                              end; (* else *)
59   96   0   2           writeln; writeln('ANOTHER MAZE? (Y or N)');
61   97   0   2           readln(InputChar)
                      until not (InputChar in ['y','Y'])
62   99   0   2   end. (* AMazeGame *)
```

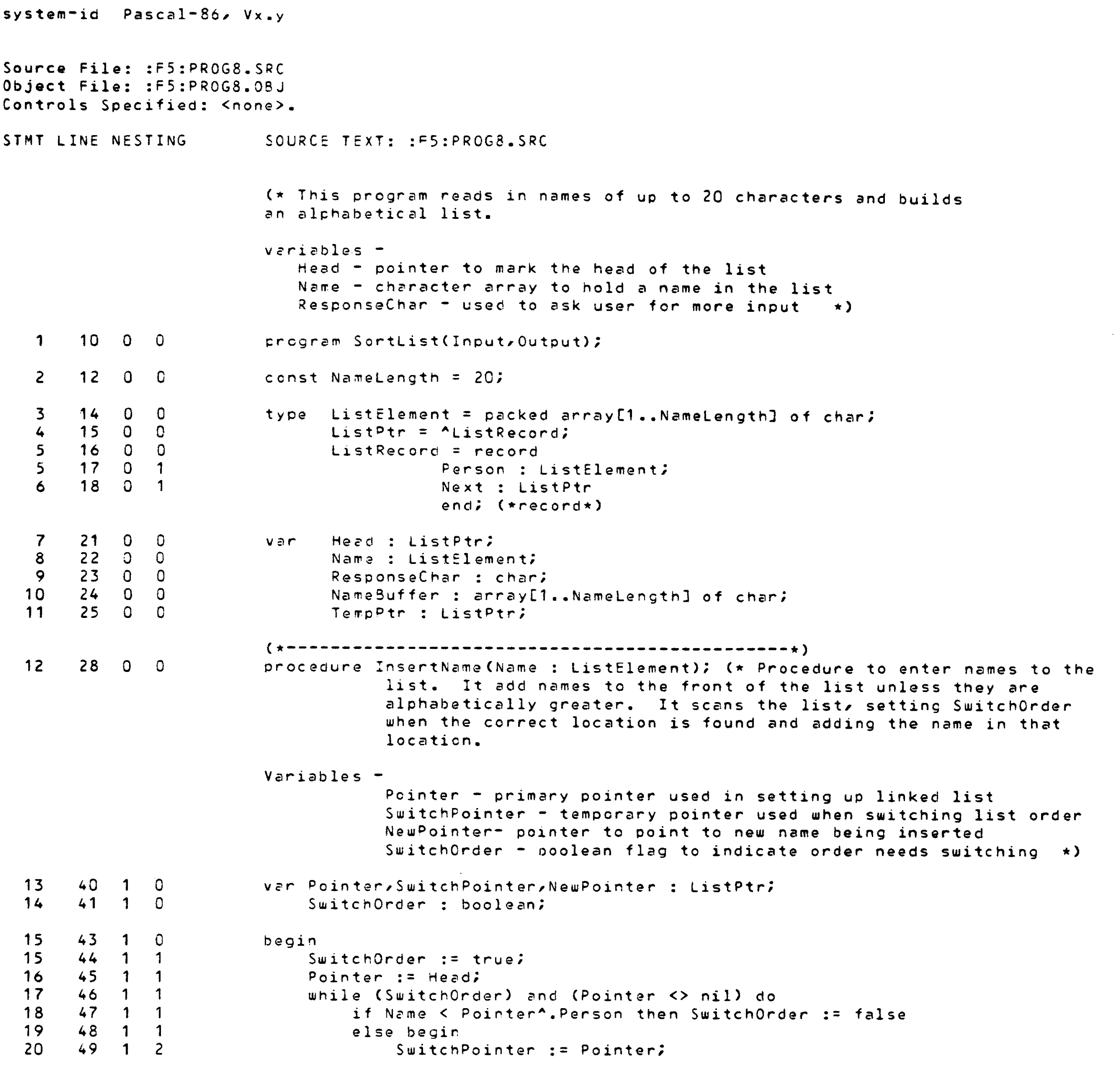
Figure 9-9. Sample Program 7: Maze Game (Cont'd.)

9.8 Sample Program 8: List Processing

The program of figure 9-10 builds a list of first names in alphabetical order as they are input. It reads in each name and inserts the name in an alphabetical linked list. It illustrates the use of dynamic variables and pointers to form a linked list.

This kind of data structure is an alternative to non-dynamic data structures like the array of records used for the tree in sample programs 2A and 2B. (Sample programs 2A and 2B could also have been implemented using dynamic variables.)

This program must be linked to the run-time support libraries (P86RN0.LIB, P86RN1.LIB, P86RN2.LIB, P86RN3.LIB), 87NULL.LIB, and any interface libraries required by your operating system.



```
system-id  Pascal-86, Vx.y

Source File: :F5:PROG8.SRC
Object File: :F5:PROG8.OBJ
Controls Specified: <none>.

STMT LINE NESTING         SOURCE TEXT: :F5:PROG8.SRC

                          (* This program reads in names of up to 20 characters and builds
                          an alphabetical list.

                          variables -
                             Head - pointer to mark the head of the list
                             Name - character array to hold a name in the list
                             ResponseChar - used to ask user for more input   *)

  1   10  0  0            program SortList(Input,Output);

  2   12  0  0            const NameLength = 20;

  3   14  0  0            type  ListElement = packed array[1..NameLength] of char;
  4   15  0  0                  ListPtr = ^ListRecord;
  5   16  0  0                  ListRecord = record
  5   17  0  1                          Person : ListElement;
  6   18  0  1                          Next : ListPtr
                                        end; (*record*)

  7   21  0  0            var   Head : ListPtr;
  8   22  0  0                  Name : ListElement;
  9   23  0  0                  ResponseChar : char;
 10   24  0  0                  NameBuffer : array[1..NameLength] of char;
 11   25  0  0                  TempPtr : ListPtr;

                          (*----------------------------------------------*)
 12   28  0  0            procedure InsertName(Name : ListElement); (* Procedure to enter names to the
                                 list.  It add names to the front of the list unless they are
                                 alphabetically greater.  It scans the list, setting SwitchOrder
                                 when the correct location is found and adding the name in that
                                 location.

                          Variables -
                                 Pointer - primary pointer used in setting up linked list
                                 SwitchPointer - temporary pointer used when switching list order
                                 NewPointer- pointer to point to new name being inserted
                                 SwitchOrder - boolean flag to indicate order needs switching  *)

 13   40  1  0            var Pointer,SwitchPointer,NewPointer : ListPtr;
 14   41  1  0                SwitchOrder : boolean;

 15   43  1  0            begin
 15   44  1  1               SwitchOrder := true;
 16   45  1  1               Pointer := Head;
 17   46  1  1               while (SwitchOrder) and (Pointer <> nil) do
 18   47  1  1                  if Name < Pointer^.Person then SwitchOrder := false
 19   48  1  1                  else begin
 20   49  1  2                     SwitchPointer := Pointer;
```

Figure 9-10. Sample Program 8: List Processing

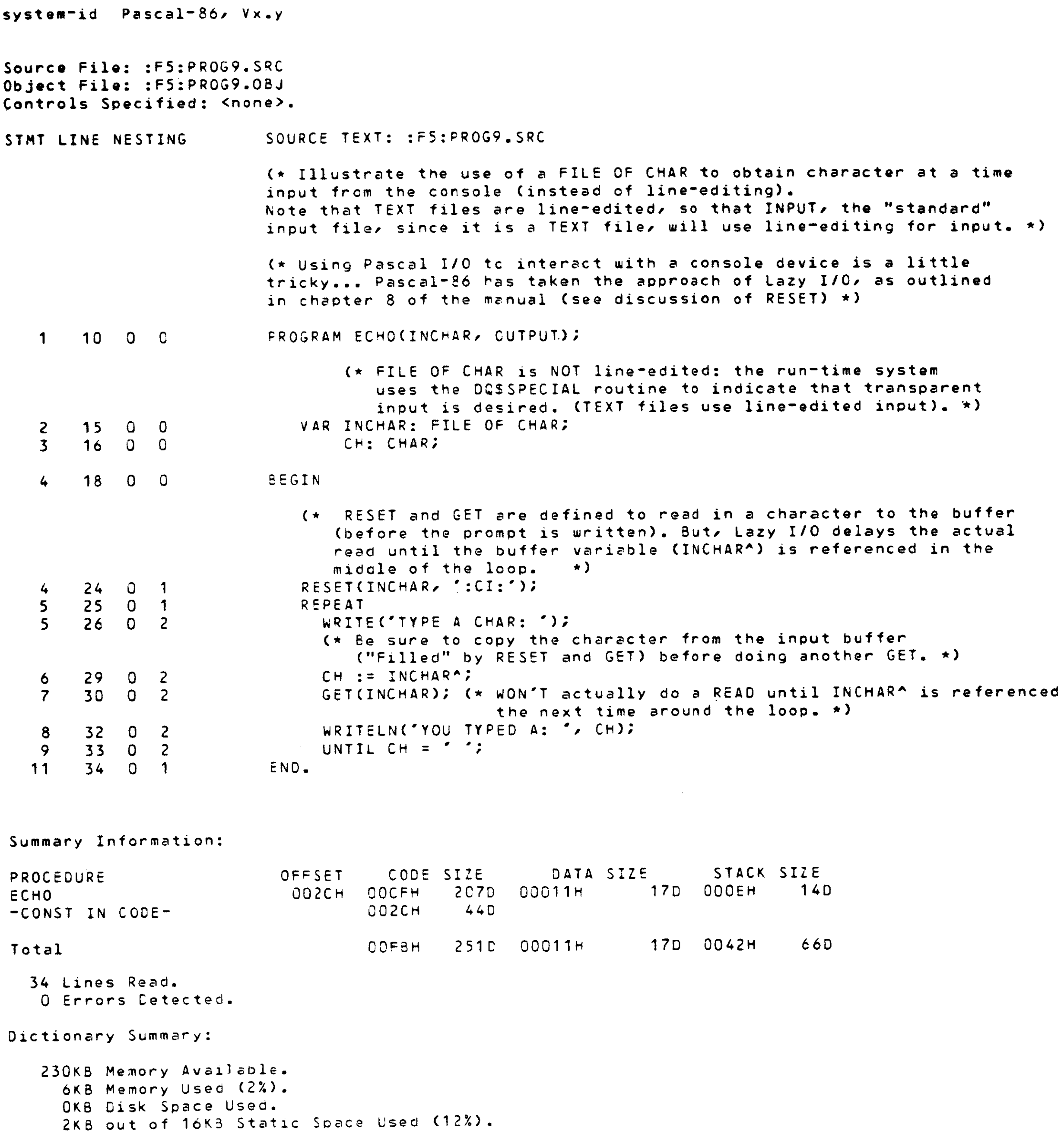
```
 21    50   1   2                Pointer := Pointer^.Next
                                 end; (* else *)
 23    52   1   1          new(NewPointer);
 24    53   1   1          NewPointer^.Person := Name;
 25    54   1   1          NewPointer^.Next := Pointer;
 26    55   1   1          if Pointer = Head then Head := NewPointer
 27    56   1   1          else SwitchPointer^.Next := NewPointer
 28    57   1   1      end;(* InsertName *)

                       (*----------------------------------------------*)

 29    61   0   0      procedure ReadName(var Name : ListElement); (* input a name *)
 30    62   1   0      var i : integer;

 31    64   1   0      begin
 31    65   1   1          for i := 1 to NameLength do NameBuffer[i] := ' ';
 33    66   1   1          for i := 1 to NameLength do
 34    67   1   1               if not eoln then read(NameBuffer[i]);
 36    68   1   1          pack(NameBuffer,1,Name);
 37    69   1   1          readln
                       end;(* ReadName *)

                       (*----------------------------------------------*)
 38    73   0   0      procedure WriteName; (* output a linked list *)

 39    75   1   0      begin
 39    76   1   1          TempPtr := Head;
 40    77   1   1          while TempPtr <> nil do begin
 41    78   1   2              writeln(TempPtr^.Person);
 42    79   1   2              TempPtr := TempPtr^.Next
                               end (* whlie *)
 43    81   1   1      end;(* WriteName *)

                       (*--------------------------------------------------*)
 44    84   0   0      begin (* SortList *)
 44    85   0   1          Head := nil;
 45    86   0   1          writeln('Begin inputting names ');
 46    87   0   1          repeat
 46    88   0   2              ReadName(Name);
 47    89   0   2              InsertName(Name);
 48    90   0   2              write('More names? ');
 49    91   0   2              readln(ResponseChar);
 50    92   0   2          until (ResponseChar in ['n','N']);
 52    93   0   1          WriteName
                       end.(* SortList *)


Summary Information:

PROCEDURE                  OFFSET    CODE SIZE       DATA SIZE       STACK SIZE
INSERTNAME                  002FH   0088H   184D                    0016H    22D
READNAME                    00E7H   0061H    97D                    0014H    20D
WRITENAME                   0148H   004CH    77D                    000EH    14D
SORTLIST                    0195H   00FFH   255D   00041H      65D  0016H    22D
-CONST IN CODE-                     002FH    47D

Total                               0294H   660D   00041H      65D  0082H   130D

  95 Lines Read.
   0 Errors Detected.

Dictionary Summary:

   230KB Memory Available.
     6KB Memory Used (2%).
     0KB Disk Space Used.
     3KB out of 16KB Static Space Used (18%).
```

Figure 9-10. Sample Program 8: List Processing (Cont'd.)

9.9 Sample Program 9: Character Input/Output

The program of figure 9-11 echoes characters that are entered at the console. It illustrates the use of FILE OF CHAR and lazy I/O to read characters one at a time from the console (instead of using line-edited TEXT files).

This program must be linked to the run-time support libraries (P86RN0.LIB, P86RN1.LIB, P86RN2.LIB, P86RN3.LIB), 87NULL.LIB, and any interface libraries required by your operating system.



```
system-id  Pascal-86, Vx.y

Source File: :F5:PROG9.SRC
Object File: :F5:PROG9.OBJ
Controls Specified: <none>.

STMT LINE NESTING       SOURCE TEXT: :F5:PROG9.SRC

                        (* Illustrate the use of a FILE OF CHAR to obtain character at a time
                        input from the console (instead of line-editing).
                        Note that TEXT files are line-edited, so that INPUT, the "standard"
                        input file, since it is a TEXT file, will use line-editing for input. *)

                        (* Using Pascal I/O tc interact with a console device is a little
                        tricky... Pascal-86 has taken the approach of Lazy I/O, as outlined
                        in chapter 8 of the manual (see discussion of RESET) *)

  1   10  0  0          PROGRAM ECHO(INCHAR, CUTPUT);

                               (* FILE OF CHAR is NOT line-edited: the run-time system
                                  uses the DQ$SPECIAL routine to indicate that transparent
                                  input is desired. (TEXT files use line-edited input). *)
  2   15  0  0             VAR INCHAR: FILE OF CHAR;
  3   16  0  0                 CH: CHAR;

  4   18  0  0          BEGIN

                           (*  RESET and GET are defined to read in a character to the buffer
                              (before the prompt is written). But, Lazy I/O delays the actual
                              read until the buffer variable (INCHAR^) is referenced in the
                              middle of the loop.   *)
  4   24  0  1            RESET(INCHAR, ':CI:');
  5   25  0  1            REPEAT
  5   26  0  2              WRITE('TYPE A CHAR: ');
                             (* Be sure to copy the character from the input buffer
                                ("Filled" by RESET and GET) before doing another GET. *)
  6   29  0  2              CH := INCHAR^;
  7   30  0  2              GET(INCHAR); (* WON'T actually do a READ until INCHAR^ is referenced
                                             the next time around the loop. *)
  8   32  0  2              WRITELN('YOU TYPED A: ', CH);
  9   33  0  2              UNTIL CH = ' ';
 11   34  0  1          END.


Summary Information:

PROCEDURE                  OFFSET    CODE SIZE       DATA SIZE       STACK SIZE
ECHO                        002CH  00CFH    207D  00011H       17D  000EH    14D
-CONST IN CODE-                    002CH     44D

Total                              00FBH    251D  00011H       17D  0042H    66D

  34 Lines Read.
   0 Errors Cetected.

Dictionary Summary:

   230KB Memory Available.
     6KB Memory Used (2%).
     0KB Disk Space Used.
     2KB out of 16KB Static Space Used (12%).
```

Figure 9-11. Sample Program 9: Character Input/Output

CHAPTER 10
COMPILER CONTROLS

You control the operation of the Pascal-86 compiler by using *compiler controls* that allow you to specify options such as inhibiting extension warning messages or generating debug records. All controls have default values preset to their common uses.

By default, the Pascal-86 compiler will produce two files: *source*.OBJ for the object module with type records, and *source*.LST for the source listing including error messages, where *source* is the filename (without extension) of the Pascal source file. If the NOEXTENSIONS control is active, the compiler will also issue warning messages when it detects an Intel extension to standard Pascal, and copy all errors and warnings to the console.

If you do not want to change these default values, you can safely skip this chapter. If you need to change any control defaults, you should read the following sections and refer to individual controls in section 10.3. A summary of these controls is provided in table 10-1.

Table 10-1. Summary of Pascal-86 Compiler Controls

Control	Abbreviation	Default	Action
CHECK/NOCHECK	CH/NOCH	NOCHECK	Check for arithmetic overflows, stack overflow, and out-of-range assignments and subscripts during compilation and run time.
CODE/NOCODE	CO/NOCO	NOCODE	Allow or prevent listing of approximate assembly code.
COND/NOCOND	None	COND	Determine whether text skipped during compilation will appear in the listing.
*DEBUG/NODEBUG	DB/NODB	NODEBUG	Genrate debug records in the object module.
EJECT	EJ	paging is automatic	Forces the start of a new page of printed output.
*ERRORPRINT/ NOERRORPRINT *ERRORPRINT(*file*)	EP/NOEP	ERRORPRINT(:CO:)	Write all compiler-generated error messages to the specified file.
*EXTENSIONS/ NOEXTENSIONS	ET/NOET	EXTENSIONS	Allow Intel extensions to standard Pascal, or (NOEXTENSIONS) issue a warning whenever the source code contains any extensions to standard Pascal.
IF/ELSEIF/ELSE/ENDIF	None	not applicable	Enable the actual conditional compilation capability by testing for conditions that are based on the value of switches.
INCLUDE(*file*)	IC	not applicable	Includes other source files as input to the compiler.
INTERRUPT(*proc* [=*n*][,...])	IT	not applicable	Designates procedures as interrupt procedures, and generates interrupt vector.
LARGE/COMPACT/SMALL	LA/CP/SM	LARGE	Determine the memory addressing techniques of a module being compilation.
LIST/NOLIST	LI/NOLI	LIST	Allow or prevent listing of source lines.

Table 10-1. Summary of Pascal-86 Compiler Controls (Cont'd.)

Control	Abbreviation	Default	Action
*OBJECT[(*file*)]/ NOOBJECT	OJ/NOOJ	OBJECT(*source*.OBJ)	Specify a filename for the object module, or prevent the creation of an object module.
*OPTIMIZE(*n*)	OT	OPTIMIZE(1)	Governs the level of optimization performed when generating object code.
*PRINT[(*file*)]/NOPRINT	PR/NOPR	PRINT(*source*.LST)	Allow or prevent printed output, or select device or file to receive printed output.
RESET/SET	None	RESET	Control the value of switches. SET establishes a value; RESET restores the value to 0.
SUBTITLE('*subtitle*')	ST	no subtitle	Put a subtitle on each page of printed output, and causes a page eject.
*SYMBOLSPACE(*n*)	SS	SS(16)	Sets the maximum amount of memory available for the compiler's internal table.
*TITLE('*title*')	TT	module name in source code	Puts a title on each page of printed output.
*TYPE/NOTYPE	TY/NOTY	TYPE	Include or omit type records in object module.
*XREF/NOXREF	XR/NOXR	NOXREF	Allow or prevent a cross-reference listing of source program identifiers.

*Primary control (all others are general).

10.1 Introduction to Compiler Controls

You can specify a control in the command line used to invoke the compiler, or in control lines that appear as part of the source file. In your specific host-system appendix, you will find information on the command line used to invoke the compiler.

When you specify a control in the invocation command line, the control remains active unless another control overrides it. A discussion of the types of controls and the rules governing them follows; specific cases of controls overriding other controls are discussed in 10.2. These cases are summarized in table 10-2.

To override the controls specified in the invocation line, controls must appear in the source file itself. For example, if the invocation line specified only the NOLIST control (all others being set to their default values), then the control to override NOLIST is its opposite form: LIST. It would appear in the source file as follows:

```
$LIST
```

A *control line* is a source line starting with a dollar sign ($) in the leftmost column. In this example, the compiler will now list the source program lines until the next occurrence, if any, of NOLIST. (All controls are described in 10.3.)

You use control lines to selectively control the compilation of sections of your source file. For example, you might want to suppress the listing of sections of your program or cause page ejects to start listings on new pages. Whenever the compiler sees a dollar sign ($) in the leftmost column of a source line, it assumes that the line is a control line, even if the dollar sign is embedded within a comment.

There are two types of controls: primary and general. *Primary controls* must occur either in the invocation command line or in a control line that precedes the first noncontrol line of the source file. You specify primary controls in the invocation line or put an initial set of primary control lines before the first source program line. You use primary controls as "global" controls that must be set before any compiling begins. These controls cannot be changed during compilation.

General controls may occur either in the invocation command line or in a control line located anywhere in the source file. These controls may be changed during compilation. General controls can override each other, but they cannot override primary controls. General control lines in the source file are considered Intel extensions to standard Pascal that cause warning messages if the NOEXTENSIONS control is active.

All controls have default values that are active unless you explicitly specify their opposite values. In typical compilations, you might not specify any controls and employ only the default values.

If the compiler detects an error caused by a primary control in the invocation line or in the initial set of primary control lines in the source file, it stops compiling and issues an error message to the console. If the compiler detects an error in general control lines (after the first set of primary control lines), it reports the error in the same manner as other compiler errors. Chapter 13 provides a discussion of all compiler error messages.

Table 10-2. Summary of the Effects of Controls on Other Controls

Control Used	Control(s) Affected or Overridden by It	Control(s) that Affect It	Control(s) that Override It
CHECK	NOCHECK	*OBJECT	*NOOBJECT NOCHECK
CODE	NOCODE	*PRINT(*file*) *OBJECT	NOCODE *NOPRINT
COMPACT	SMALL, LARGE		*NOOBJECT
COND	NO COND		NOLIST *NOPRINT
*DEBUG	*NODEBUG	*OBJECT(*file*)	*NODEBUG *NOOBJECT

*EJECT	*TITLE('*title*')
	SUBTITLE('*subtitle*')
		*PRINT(*file*)	*NOPRINT
			NOLIST

*ERRORPRINT(*file*)	*ERRORPRINT(:CO:) *NOERRORPRINT		*NOERRORPRINT
*NOERRORPRINT	*ERRORPRINT(*file*)		*ERRORPRINT(*file*) *NOPRINT

*NOEXTENSIONS	*EXTENSIONS	*ERRORPRINT(*file*)
		*NOERRORPRINT
		*PRINT(*file*)
			*EXTENSIONS

INCLUDE(*file*)			
INTERRUPT(*proc* [,...])		*NOOBJECT	
LARGE	SMALL, COMPACT		*NOOBJECT
NOLIST	LIST SUBTITLE('*subtitle*') EJECT	*PRINT(*file*)	LIST *NOPRINT

*OBJECT(*file*)	*OBJECT(*source*.OBJ)
	INTERRUPT(*proc* [,...])
	*DEBUG
	*TYPE
			*NOOBJECT

Table 10-2. Summary of the Effects of Controls on Other Controls (Cont'd.)

Control Used	Control(s) Affected or Overridden by It	Control(s) that Affect It	Control(s) that Override It
*NOOBJECT	*OBJECT(_file_) INTERRUPT(_proc_ [,...]) *DEBUG *TYPE CHECK CODE		*OBJECT(_file_)
*OPTIMIZE(_n_)		*NOOBJECT	
*PRINT(_file_)	CODE *NOEXTENSIONS NOLIST EJECT TITLE('_title_') SUBTITLE('_subtitle_') PRINT(_source_.LST)		*NOPRINT

*NOPRINT	*PRINT(_file_)
	*PRINT(_source_.LST)
	CODE
	EJECT
	*NOERRORPRINT
	*NOEXTENSIONS
	NOLIST
	TITLE('_title_')
	SUBTITLE('_subtitle_')
	XREF
			*PRINT(_file_)
			*PRINT(_source_.LST)

SMALL	COMPACT, LARGE		*NOOBJECT
SUBTITLE('_subtitle_')		NOLIST EJECT PRINT(_file_)	*NOPRINT
*SYMBOLSPACE(_n_)			
*TITLE('_title_')		NOLIST PRINT(_file_)	*NOPRINT
*TYPE	*NOTYPE	*OBJECT(_file_)	*NOOBJECT *NOTYPE
*XREF	*NOXREF	*PRINT(_file_)	*NOXREF *NOPRINT

*Primary control (all others are general).

10.2 Using Controls

Controls to the compiler govern the format, processing, and content of both the input source file(s) and the output file(s). Certain controls in their default forms override other controls that are explicitly stated. This section describes the use of controls according to the areas they govern, and suggests which controls should be used during specific stages of program development.

In the following sections, an asterisk (*) denotes a primary control (or control pair). All other controls are general controls.

10.2.1 Listing Device or File Selection

The following controls govern the selection of the device or file to receive compiler listings and error/warning messages:

```
*PRINT[(file)]/NOPRINT
*ERRORPRINT[(file)]/NOERRORPRINT
```

Use the PRINT control to select the device or file to receive *all* printed output. Under NOPRINT, *only error messages* will be output to either the console or the ERRORPRINT file, which you select with the ERRORPRINT control.

The NOPRINT control overrides all of the listing format controls described in 10.2.2, since it governs all printed output. You can, however, select a different file or device to receive error messages. Even if the NOPRINT control is active, error messages will always appear somewhere—either in a different file specified in an ERRORPRINT control, or at the console if the ERRORPRINT control is not specified (or if NOERRORPRINT is specified).

To generate a listing that includes error messages and the complete (or partial) source listing (as governed by format controls discussed below), use the PRINT control to specify the listing file, or allow the default PRINT control to send the listing to *source*.LST. If you select an ERRORPRINT file, error messages will appear twice: once in the ERRORPRINT file, and once in the listing file governed by the PRINT control.

10.2.2 Controlling Listing Format and Content

If PRINT is active, the following controls govern the format and content of printed output. The default value of a control pair is listed first:

```
 NOCODE/CODE
 EJECT
*EXTENSIONS/NOEXTENSIONS
 LIST/NOLIST
 SUBTITLE('subtitle')
*TITLE('title')
*NOXREF/XREF
 COND/NOCOND
```

The default values allow listing of the source program without the approximate assembly code listing (NOCODE), without the identifier cross-reference (NOXREF), and without any extension warning messages (EXTENSIONS).

These default values assume the general case. If you need the approximate assembly code listing of portions of the source file, use the CODE control. If you need to suppress certain portions of the source listing, use NOLIST. Note that the NOLIST control does *not* override the CODE control.

Under the default EXTENSIONS, extension warning messages are not issued (i.e., Intel extensions to standard Pascal are accepted without any warnings). If you specify NOEXTENSIONS, extension warning messages will occur if the compiler detects an Intel extension to standard Pascal. These warnings are directed to the file governed by PRINT and ERRORPRINT, and they do not stop the compilation process. Deviations that are *not* supported by Pascal-86 generate real errors. Control lines in the source file (after the initial set of primary controls) are considered Intel extensions to standard Pascal; under NOEXTENSIONS, they cause extension warnings.

The XREF control directs the compiler to produce a symbol and identifier cross-reference, as described in 11.1.5 and in 10.3. NOXREF (the default) suppresses this action, and NOPRINT overrides XREF.

Although paging is automatic (every 60 lines), you can force a page eject on any line by using the EJECT control. An EJECT in a control line is ignored if the control line occurs in an area governed by the NOLIST control. TITLE and SUBTITLE controls specify titles and subtitles in the listing. If NOLIST is in effect, the subtitle specified is saved until listing resumes with the LIST control. All of these controls are ignored if NOPRINT is active.

Conditional compilation allows two listing options: COND and NOCOND. COND specifies that any skipped source code will be listed (without statement or level numbers) and NOCOND specifies that skipped text will not be listed.

10.2.3 Source Selection and Processing

The following controls govern the selection and processing of source files. The default value of each pair is listed first.

```
*EXTENSIONS/NOEXTENSIONS
 INCLUDE(file)
```

Pascal-86 allows only one primary source file, but other source files may be included in the compilation by specifying them in INCLUDE controls. For instance, you can save the PUBLIC sections common to several modules in a separate source file, and you can INCLUDE this file whenever the PUBLIC sections it contains are needed by the modules being compiled.

The INCLUDE control must be the rightmost (last) control on a source control line. The compiler issues a nonfatal error message when controls are placed to the right of the INCLUDE control.

The EXTENSIONS control (default value) allows Intel extensions to standard Pascal to pass through compilation without generating warning messages. NOEXTENSIONS directs the compiler to check for these extensions and to issue warnings. Use NOEXTENSIONS for programs that you want to conform to standard Pascal.

10.2.4 Conditional Compilation

The following controls pertain to conditional compilation, a process that allows the compiler to skip selected sections of the source file if specified conditions are not met. Likewise, conditional compilation can be used to select various compiler controls by testing specified conditions in the source code. The default value of each control pair is listed first.

```
IF/ELSEIF/ELSE/ENDIF
RESET/SET
COND/NOCOND
```

IF, ELSEIF, ELSE, and ENDIF are general controls that enable the actual conditional compilation process. An IF control and an ENDIF control delimit an IF element. An IF element has several different forms; the most complete form includes one or more ELSEIFs, followed by an optional ELSE. The operands in an IF element are not type-checked; they must be either a byte constant or predefined in a RESET/SET control.

RESET and SET are general controls that determine the value of various switch assignments that can be used in a limited way to form conditions. These conditions are then tested by the IF and ELSEIF controls to determine the value of the least significant bit. Based on the results of this test, the compiler determines which sections of code should be compiled. The value of the switch assignment may be any whole number constant from 0 to 255. SET establishes the value; RESET restores the value to 0.

COND/NOCOND determines whether text skipped during conditional compilation will appear in the listing. The COND control specifies that any skipped text will be listed (without statement or level numbers). A COND control cannot override a NOLIST or LIST control. Also note that a COND control is not processed if it is in a portion of skipped text. The NOCOND control specifies that skipped text will not be listed.

Conditional compilation has many useful applications. For example, conditional compilation can be used when porting a program to different architectures, or when a program contains several features that are not required for each implementation. Rather than writing a separate program for each case, you can write one program that uses conditional compilation to select the necessary sections of code for each application.

10.2.5 Object Content and Program Checkout

The following controls govern the selection and content of the object module, and implement program checking. The default value of a control pair is listed first.

```
*OBJECT(file)/NOOBJECT
*NODEBUG/DEBUG
*TYPE/NOTYPE
 NOCHECK/CHECK
```

The OBJECT control selects a file to receive the object module. The default file has the same root name as the source file, with the extension OBJ (i.e., if PROG1.SRC is the source file, PROG1.OBJ becomes the object file). NOOBJECT prevents the generation of an object module and directs the compiler to perform only a quick syntax and semantic check of the source file. It also inhibits the execution of the OBJECT phase, overriding CODE and CHECK.

The DEBUG control generates debug records in the object module that are used by symbolic debuggers such as PSCOPE, DEBUG-86, and the ICE-86 emulator. The default value NODEBUG suppresses the generation of debug records. NOOBJECT overrides DEBUG. The DEBUG control generates debug records and does not affect any program checkout features.

The TYPE control (default value) generates type records in the object module that are used for type checking by the linker. TYPE is the default, because type checking is one of the advantages of using Pascal. Type records provide the mechanism for enforcing type compatibility between separately-compiled modules. TYPE information is also used by PSCOPE to display or modify memory variables. The NOTYPE control suppresses the generation of type records. NOOBJECT overrides TYPE.

The CHECK control directs the compiler to check for out-of-range assignments, out-of-bounds array subscripts, stack overflow, and integer overflow. If possible, violations are checked at compile time, but some run-time checking may be required. NOCHECK suppresses all checking activity. NOOBJECT overrides the compile-time checking performed by CHECK.

10.2.6 Program Optimization and Run-Time Environment

The following controls optimize code and memory size requirements and affect the run-time environment of a particular program. The default value of the control is listed first.

```
*OPTIMIZE(n)
 LARGE/COMPACT/SMALL
 INTERRUPT(proc[=n][,...])
```

The OPTIMIZE control allows you to specify the level of optimization you want the compiler to perform when it generates object code. Two optimization levels are provided: OPTIMIZE(1) (the default), and OPTIMIZE(0). OPTIMIZE(0) performs more limited optimization and is recommended when debugging programs with PSCOPE or DEBUG-86.

The LARGE, COMPACT, and SMALL controls determine the memory addressing techniques of a given object module. If you are unsure of the memory requirements of your program, LARGE (the default) is recommended because it directs the compiler to make no special assumptions. You can improve code efficiency by using the smallest control possible, given your program's specific run-time memory requirements. Note that every module in a program must be compiled with the same control. (Appendix I outlines extensions to these controls.)

The INTERRUPT control enables you to compile specific procedures as interrupt procedures. Dynamic interrupt number assignments (set by the SETINTERRUPT procedure) within the source program override the assignments made in the INTERRUPT control. If an interrupt occurs during run time, and is associated by number with an interrupt procedure, the interrupt procedure gains control. (Interrupt handling is described in detail in Appendix K.)

10.2.7 Use of Controls in Stages of Development

When you are compiling a program for the first time, use the default control settings with the following exceptions:

- Use CHECK, OPTIMIZE(0), and DEBUG for program checkout; then use PSCOPE, DEBUG-86, or ICE-86A for symbolic program debugging.
- Use XREF to generate a symbol and identifier cross-reference to aid your debugging and maintenance efforts.

Definitions of PUBLIC procedures and functions (needed for an interface specification that is common to several modules) can be maintained in a separate file and included with the source file by using the INCLUDE control. (For information on interface specifications, see 4.2.3.)

For quick compiling and error reporting, you can maximize compile speed by using default settings for all but one control; use NOPRINT to supress printed output. (Errors will be reported at the console, or use ERRORPRINT to redirect errors to a file.)

When preparing programs to test with the ICE-86A or ICE-88 emulator, use the CODE control during compilation to list the pseudo-assembly instructions and addresses. The CODE control can help you recode certain portions in assembly language.

Use the NOLIST control to save time by not listing portions of the source code that are already debugged. To make your listing more readable, use EJECT, TITLE, and SUBTITLE. You can direct the final listing to a specific output file using the PRINT control, and direct the final object module to a specific output file using the OBJECT control.

To enforce compatibility with the Pascal standard, use the NOEXTENSIONS control early in program development to generate warning messages whenever the compiler encounters an Intel extension to standard Pascal. You can also use the CODE control to help recode non-standard areas in assembly language.

10.3 Descriptions of Individual Controls

NOTE

Sample invocation lines for most compiler controls are provided on a foldout page in your specific operating-system appendix.

10.3.1 CHECK/NOCHECK

Checks for invalid references, overflow, and out-of-range assignments and subscripts during compilation or at run time.

Syntax

```
CHECK
NOCHECK
```

Abbreviation

```
CH/NOCH
```

Default

```
NOCHECK
```

Type

General

Description

The CHECK control provides a way to check for certain violations during compilation and at run time. The compiled code checks for the following:

- Array subscripts out of bounds
- Stack overflow
- Overflow in integer computations
- Out-of-range assignments

Example

```
$CHECK
```

In the above control line, the CHECK (abbreviated CH) control causes the subsequent code to implement checking.

NOTE

> Violations detected at run-time cause the execution of an error handler. Overflow in integer computations generates an Interrupt 4, which in turn invokes the trap handler in effect at the time of the interrupt. Appendix K provides information about run-time support and interrupts.

10.3.2 CODE/NOCODE

Allows or prevents the listing of approximate assembly code.

Syntax

```
CODE
NOCODE
```

Abbreviation

```
CO/NOCO
```

Default

```
NOCODE
```

Type

General

Description

The CODE control directs the compiler to produce a listing of the approximate assembly code for the generated object code (in a form that resembles 8086 assembly language). This listing occurs only for portions of the source code where the CODE control is active—it stops when a NOCODE is encountered. No code listing is generated if the NOOBJECT control is active. The approximate assembly code listing is appended to the source listing in the listing file created by the PRINT control (see PRINT/NOPRINT).

The NOCODE control prevents the listing of the approximate assembly code. The default control setting (if you specify neither control) is NOCODE.

Example

```
$CODE
```

The CODE control in this control line lists the approximate assembly code for the object code and appends the listing to the source listing.

```
$NOCODE
```

The NOCODE control in this control line turns off the action of the CODE control.

NOTE

The CODE control cannot create printed output if the NOPRINT control is in effect. Section 11.1.5 gives a sample listing of approximate assembly code.

10.3.3 COMPACT

Specifies memory addressing techiniques of a program under compilation.

Syntax

```
1.  COMPACT{[(-CONST IN DATA-)]}
           { (-CONST IN CODE-)  }

2.  COMPACT ([subsystem-id][submodel]EXPORTS public-list[; EXPORTS public list]...)

3.  COMPACT (subsystem-id[submodel] HAS module-list [{; HAS module-list    }]...)
                                                    [{; EXPORTS public-list}]
```

Abbreviation

CP

Default

LARGE (see 10.3.12)

Type

General

Description

The COMPACT control directs the compiler to perform certain memory addressing optimizations that help reduce the amount of code produced. If you do not need to optimize your code size, or if you are not sure that your program meets the COMPACT memory restrictions, simply use the default LARGE control. (For more advanced methods of memory optimization, see Appendix I.)

This section discusses the COMPACT control in its simplest form (see (1) above). The syntax in (2) and (3) applies only to the extended controls, which are discussed in Appendix I. Also see Appendix I for placement of these controls.

Modules compiled with the COMPACT control have four sections: code, constant, data, and stack (see 11.2). When these modules are linked, similar sections from each module are combined to form segments. A COMPACT program has three segments: code, data, and stack.

In the default COMPACT case (—CONST IN DATA—), the code sections from all modules are allocated space in one segment, which is addressed relative to the CS register. All constant and data sections (excluding dynamic variables) are combined in a second segment, which is addressed relative to the DS register. The stack, containing parameters and local variables, is addressed relative to SS.

If (—CONST IN CODE—) is specified, the code and constant sections from all the modules are allocated space within one segment, which is addressed relative to the CS register. The data sections (excluding dynamic variables) are combined in a second segment, which is addressed relative to the DS register. The stack is addressed relative to SS.

The maximum size, each, of the code segment (including constants if —CONST IN CODE—), the data segment (including constants if —CONST IN DATA—), and the stack segment, is 64K.

Dynamic variables are allocated on the heap, which is outside of the three segments discussed above, and are addressed with 32-bit pointers. The maximum storage for dynamic variables is one megabyte.

References to any location require only a 16-bit offset, using these segment addresses. Since the code, data, and stack segments are fully defined by the time the program is loaded, the addresses in the CS, DS, and SS registers are never changed.

10.3.4 COND/NOCOND

These controls determine whether text skipped during compilation will appear in the listing.

Syntax

```
COND
NOCOND
```

Abbreviation

none

Default

```
COND
```

Type

General

Description

The COND control specifies that any skipped text will be listed (without statement numbers or level numbers). Note that a COND control cannot override a NOLIST or NOPRINT control. Also note that a COND control is not processed if it is in a portion of skipped text.

The NOCOND control specifies that skipped text will not be listed. However, controls that delimit the skipped text are listed, indicating that text has been skipped. Again, note that a NOCOND control is not processed if it is in a portion of skipped text.

10.3.5 DEBUG/NODEBUG

Generates debug records in the object module.

Syntax

```
DEBUG
NODEBUG
```

Abbreviation

```
DB/NODB
```

Default

```
NODEBUG
```

Type

Primary

Description

The DEBUG control generates debug records which contain the name and relative address of each symbol whose address or stack frame offset is known by the compiler, and the statement number and relative address of each source statement. The DEBUG control generates debug records in the object module—it does not imply the CHECK control checkout features.

The default setting, NODEBUG, prevents the generation of these records.

Example

```
$DEBUG
```

The DEBUG control in this control line generates debug records in the object module.

NOTE

The DEBUG control is ignored if the NOOBJECT control is in effect.

10.3.6 EJECT

Forces the start of a new page of printed output.

Syntax

```
EJECT
```

Abbreviation

```
EJ
```

Default

paging is automatic

Type

General

Description

The EJECT control terminates the printing of the current page and starts a new page. The control line containing the EJECT control is the last line printed on the old page.

If you do not use the EJECT control, a page eject will occur automatically after every 60 lines.

Example

```
$EJECT
```

The EJECT control in this control line forces the start of a new page, after this control line is printed.

NOTE

The EJECT control is ignored if the NOLIST or NOPRINT controls are in effect.

10.3.7 ERRORPRINT/NOERRORPRINT

Copies all compiler-generated error messages to the specified file.

Syntax

```
ERRORPRINT[(file)]
NOERRORPRINT
```

Abbreviation

```
EP/NOEP
```

Default

```
ERRORPRINT(:CO:)
```

Type

Primary

Description

The ERRORPRINT control with the optional *file* argument directs all compile-time error messages to both the *file* specified *and* the listing file (file specified in a PRINT control, or *source*.LST by default), if the listing file is not suppressed by NOPRINT. If the listing file is suppressed by the NOPRINT control, errors appear only in the *file* specified. You must supply a legal pathname for *file* or an error will occur.

The ERRORPRINT control without the *file* argument (i.e., the default setting ERRORPRINT) directs all compile-time error messages to the console (:CO:) *and* thè listing file, if the listing file is not suppressed by NOPRINT. In other words, the default argument for *file* in an ERRORPRINT control is :CO: for the console.

The NOERRORPRINT control directs all compile-time error messages to the *listing file only*, i.e., the file specified in a PRINT control; or *source*.LST, the default listing file (see the PRINT control). If the listing file is suppressed by NOPRINT, the NOERRORPRINT control is ignored, and all compile-time error messages appear at the console.

If the maximum number of open files allowed by the system is allocated, and the ERRORPRINT control specifies a disk file, then the compiler will close the ERRORPRINT file. If an additional file needs to be opened, and this occurs, the ERRORPRINT disk file will no longer be updated and a message will be printed on the last line of that file. NOERRORPRINT is then in effect for the duration of the compiler.

Example

```
$NOERRORPRINT
```

The NOERRORPRINT control in this control line sends error messages to the listing file only, or to the console if the listing file is suppressed by NOPRINT.

NOTE

Even if you use NOERRORPRINT and NOPRINT, error messages still appear at the console. Without NOPRINT, NOERRORPRINT sends error messages only to the listing file.

10.3.8 EXTENSIONS/NOEXTENSIONS

Allows Intel extensions to standard Pascal, or issues an extension warning whenever the source program contains any nonstandard Pascal feature.

Syntax

```
EXTENSIONS
NOEXTENSIONS
```

Abbreviation

```
ET/NOET
```

Default

```
EXTENSIONS
```

Type

Primary

Description

The NOEXTENSIONS control directs the compiler to check for any Pascal-86 features in the source program that are Intel extensions to standard Pascal as defined in the ANSI/IEEE770X3.97–1983. Whenever the compiler finds such a feature, it issues an extension warning message as an error message, but it continues to compile the program and produce the object module. The compiler also issues an extension warning for any control line that occurs after the initial set of primary controls.

The EXTENSIONS control allows legitimate Intel extensions to be processed without warning messages. These controls do not adversely affect the compilation.

Example

```
$NOEXTENSIONS
```

The NOEXTENSIONS control in this control line causes extension warnings to occur for any Intel extension to standard Pascal.

NOTE

> Extension warning messages are treated as compile-time error messages, and are sent to the appropriate file or to the console depending on the setting of the ERRORPRINT and NOPRINT controls (see ERRORPRINT).

10.3.9 IF/ELSEIF/ELSE/ENDIF

These controls allow the actual conditional compilation capability. They cannot be used to invoke the compiler. Each control must be placed on a separate control line, as explained in 10.1.

Syntax

1. The simplest form of an IF element is as follows:

```
$ IF condition<cr>
text
$ ENDIF
```

where

condition<cr> is a limited form of Pascal expression in which the only operators allowed are OR, NOT, AND, $<$, $\leq$, $=$, $\geq$, $<>$, and $>$. The only operands allowed are switches and whole number constants from 0 to 255. If the switch does not appear in a previously defined SET control, a value of false(0) is assumed. Parenthesized subexpressions are allowed. In these restrictions, condition is evaluated as defined by standard Pascal. Note that condition must be followed by a carriage return.

text is *text* that is processed normally by the compiler if the least significant bit of the value of condition is a 1, and is *text* that is skipped if the least significant bit is a 0. *text* may contain a combination of source and compiler controls. Note that when the compiler skips *text*, compiler controls in that portion are not processed.

2. The second form of an IF element contains an ELSE element:

```
$ IF condition
text1
$ ELSE
text2
$ ENDIF
```

In this construction, *text1* is processed normally if the least significant bit of the value of condition is a 1, and *text2* is skipped. If the bit is a 0, *text2* is processed normally and *text1* is skipped.

An IF element may contain only one ELSE element.

3. The most general form of IF element allows one or more optional ELSEIF elements before the ELSE element:

```
$ IF condition1
text1
$ ELSEIF condition2
text2
$ ELSEIF condition3
text3
.
.
.
$ ELSEIF condition
textn-1
```

```
$ELSE
textn
$ENDIF
```

ELSEIF and ELSE elements are optional.

The conditions in an IF element are tested sequentially. As soon as a condition yields a value with 1 as its least significant bit, the corresponding text is processed normally. All other text referenced in the IF element is skipped. If none of the conditions yield a least significant bit of 1, text in the ELSE element is processed normally and all other text in the IF element is skipped.

Abbreviation

none

Default

not applicable

Type

General

Description

The IF/ELSEIF/ELSE/ENDIF controls enable conditional compilation by testing for conditions based on the value of switches. An IF control and an ENDIF control delimit an IF element. An IF element has several different forms; the most complete form includes one or more ELSEIFs, followed by an optional ELSE. These operands are not type-checked, and they must be either a byte constant or predefined in a RESET/SET control.

Primary controls may not be used in conditional compilation blocks.

Example

```
$IF S1
      traceMsg1(TRUE, newval);
      valctr:=valctr + 1;
$ELSEIF S2
      TraceMsg2(TRUE, newval, oldval);
      Abort;
$ELSE
      link(newval)
$ENDIF
```

Here, S1 and S2 are switches that control printing of information for the programmer's use. To print both newval and oldval at run-time, set S2 when the compiler is invoked.

```
PASC86 pathname SET (S2)
```

10.3.10 INCLUDE

Includes other source files as input to the compiler.

Syntax

```
INCLUDE(file)
```

Abbreviation

```
IC
```

Default

no included files

Type

General

Description

When the compiler encounters the INCLUDE control in the source file, it reads from the other source file named by *file*, until it reaches the end of the *file*. Then the compiler resumes reading the source lines that follow the INCLUDE control line in the original source file.

Example

```
$INCLUDE(PUBLIC.SRC)
```

Read source lines from the file PUBLIC.SRC.

NOTE

The INCLUDE control must be the rightmost control in a control line (or the only control in that line).

The included file may itself contain INCLUDE controls, but the nesting of included files cannot exceed five (five included files).

The compiler always forces an end-of-line after reading from an included file.

Your *file* must be a valid pathname, or an error will occur.

10.3.11 INTERRUPT

Designates procedures as interrupt procedures, and generates the interrupt vector.

Syntax

INTERRUPT(*procedure*[=*number*][,*procedure*[=*number*]]..)

Abbreviation

IT

Default

no interrupt procedures

Type

General

Description

The INTERRUPT control allows you to specify procedures to be compiled as 8086 interrupt procedures, and to generate an interrupt vector.

The *procedure* you supply is the identifier for the procedure to be compiled as an interrupt procedure. You can optionally specify an equal sign and a *number* for each *procedure*, and you can specify multiple *procedure*, as well as multiple INTERRUPT controls. The *number* is the number of the interrupt to be associated with the specified *procedure*; this number must be in the range 0 to 255, or an error will occur. You can only specify one *procedure* for each *number*.

When you include *number*, the compiler creates an interrupt vector consisting of a 4-byte entry for the interrupt procedure. For interrupt number *n*, the entry for the interrupt procedure is located at absolute memory location *n* times four.

Examples

```
$INTERRUPT(INT1=1, INT2=2, INT3=3)
```

This control line specifies procedure INT1 as an interrupt procedure for interrupt 1, at location 4; INT2 for interrupt 2, at location 8; and INT3 for interrupt 3, at location 12.

NOTE

The *procedure* specified must appear at the outer level of nesting (i.e., nested only in the program block), without any parameters. The *procedure* may not be passed as a procedural parameter.

After the program is loaded, a *procedure* is executed whenever the 8086 interrupt associated with the *procedure* occurs. Section K.1 provides more information about run-time interrupt processing, and section 8.9 describes the predefined interrupt control procedures.

If you use the SETINTERRUPT procedure within your program, at run time the SETINTERRUPT procedure's interrupt number assignment will take precedence and override the INTERRUPT control assignment. The SETINTERRUPT procedure is described in 8.9.1.

10.3.12 LARGE

Specifies memory addressing techniques of a program under compilation.

Syntax

```
1.  LARGE  { [(-CONST IN CODE-)] }
           {  (-CONST IN DATA-)  }

2.  LARGE  ([subsystem-id][submodel]EXPORTS public-list[; EXPORTS public-list]...)

3.  LARGE  ( subsystem-id[submodel] HAS module-list [{ ; HAS module-list     }] ... )
                                                   [{ ; EXPORTS public-list }]
```

Abbreviation

LA

Default

LARGE

Type

General

Description

The LARGE control provides the simplest form of memory addressing. Unlike SMALL and COMPACT (10.3.3 and 10.3.19), it does not put strict limits on the amount of code, constant, data, and stack space available within a program. It also does not optimize a program's storage space and the data references between program modules. If this optimization is required, see 10.3.3 and 10.3.19. (For more advanced methods of program optimization, see Appendix I.)

This section discusses the LARGE control in its simplest form (see (1) above). The syntax in (2) and (3) applies only to the extended controls, which are discussed in Appendix I. Also see Appendix I for placement of these controls.

Modules compiled with the LARGE control have four sections: code, constant, data, and stack (see 11.2). In the default LARGE case (—CONST IN CODE—), the code and constant sections from each module are not combined; they make up their own segment. Consequently, the maximum size of the code segment for each module is 64K, making the total storage available for all code segments in the program greater than 64K.

The data sections (excluding dynamic variables) from each module are also not combined, and make up their own segments. These segments include the constant sections if (—CONST IN DATA—) is specified. A module that requires more than 64K for data storage may be placed in more than one data segment.

At any moment during program execution, one code segment and one data segment are "current." These segments are paired so that the current code and data segments are always from the same module. During program execution, the segment addresses for the current code and data segments are kept in the CS and DS registers, respectively. They are updated whenever a public procedure is activated, as new code and data segments may have to be loaded to access public information.

The stack sections from all modules, containing parameters and local variables, are combined in one segment, which is addressed relative to the SS register. The maximum size of the stack is 64K.

Dynamic variables are allocated on the heap and are addressed with 32-bit pointers. The maximum storage for dynamic variables is one megabyte.

10.3.13 LIST/NOLIST

Allows or prevents the listing of source lines.

Syntax

```
LIST
NOLIST
```

Abbreviation

```
LI/NOLI
```

Default

```
LIST
```

Type

General

Description

The LIST control directs the compiler to resume (or begin) listing the program with the next source line read. The NOLIST control directs the compiler to stop listing the program until the next occurrence, if any, of a LIST control. If you specify neither, the compiler will continue to create a listing of the program (the default is LIST).

Example

```
$LIST
```

This control line starts the listing of source lines with the next line read.

NOTE

> When you specify neither, or when LIST is in effect, all lines from the source file (or from an included file), including control lines, are listed. When NOLIST is in effect, only source lines associated with error messages are listed.
>
> Note that the LIST control cannot create a listing if the NOPRINT control is in effect.
>
> The NOLIST control does *not* override the CODE control.

10.3.14 MOD86/MOD186

MOD86 and MOD186 are primarily controls that direct the compiler to generate optimized code for the 8086 and 80186 processors, respectively.

Syntax

```
MOD86
MOD186
```

Abbreviation

none

Default

```
MOD86
```

Type

Primary

Description

MOD86 specifies that the object module includes instructions for execution on the 8086 processor.

The MOD186 control allows the compiler to generate an extended set of instructions in the object module for use on the 80186 processor.

10.3.15 OBJECT/NOOBJECT

Specifies a filename for the object module, or prevents the creation of an object module.

Syntax

```
OBJECT[(file)]
NOOBJECT
```

Abbreviation

```
OB/NOOJ
```

Default

```
OBJECT(source.OBJ)
```

Type

Primary

Description

The OBJECT control directs the compiler to produce an object module. You can optionally specify a *file* for this object module by providing a legal pathname (filename with optional device specifier) for *file*.

If you do not specify a *file*, or if you do not use the OBJECT control, the compiler will still produce the object module and direct it to the same disk or device as the source file, using the filename *source*.OBJ (where *source* is the root name of the source file).

The NOOBJECT control prevents the creation of an object module, and directs the compiler to perform only a quick syntax and semantic check of the source file. It also inhibits the execution of the OBJECT phase, overriding the CODE and CHECK controls.

Example

```
$OBJECT(TEMP.OBJ)
```

This control line directs the compiler to put the object module in the file TEMP.OBJ.

NOTE

Section 11.2 provides details on the object module sections.

10.3.16 OPTIMIZE

Governs the level of optimization performed in generating object code.

Syntax

```
OPTIMIZE(0)
OPTIMIZE(1)
```

Abbreviation

```
OT
```

Default

```
OPTIMIZE(1)
```

Type

Primary

Description

The OPTIMIZE(0) control directs the compiler to turn off object code optimization between program lines. This limited optimization is required for programs being debugged under PSCOPE (the high-level language debugger) and is recommended when using DEBUG-86.

The OPTIMIZE(1) control directs the compiler to perform code optimizations both within and between program statements.

Example

```
$OT(0)
```

The OPTIMIZE(0) (abbreviated OT(0)) control in this control line turns off code optimization between program statements.

NOTE

> The OPTIMIZE control will not produce code if the NOOBJECT control is active.

10.3.17 PRINT/NOPRINT

Allows or prevents printed output, or selects the device or file to receive printed output.

Syntax

```
PRINT[(file)]
NOPRINT
```

Abbreviation

```
PR/NOPR
```

Default

```
PRINT(source.LST)
```

Type

Primary

Description

The PRINT control directs the compiler to produce printed output (listings), and the NOPRINT control stops the compiler from producing printed output. If you specify neither, the compiler will produce listings and put them in a file that has the same name as the source input file, but with an LST extension. This new LST file will be created on the same device used for the source file. For example, if your source file is named SOURCE and you use neither control, or use only the simple PRINT control (the default), the compiler will create the listing as SOURCE.LST.

If you specify a PRINT control with a *file* in parentheses, the compiler will put the listings in the file or device named by *file*, which must be a legal pathname for a file or device.

Example

```
$PRINT(:LP:)
```

This control line sends printed output to the line printer.

NOTE

> If you specify the NOPRINT control, the compiler will not produce listings—even if you specify other controls such as LIST or CODE. When the NOPRINT control is in effect, the compiler will not produce any printed output except error messages.

10.3.18 RESET/SET

RESET and SET control the value of switches. These values are used as test conditions during conditional compilation.

Syntax

The RESET control sets the value of each switch to false (0), and has the following form:

RESET (*switch list*)

where

switch list contains one or more switch names that have already been used in SET controls.

Abbreviation

none

Default

RESET (0)

Type

General

Syntax

The simplest form of the SET control is as follows:

SET (*switch assignment list*)

where

switch assignment list consists of one or more switch assignments separated by commas.

A switch assignment has the following form:

switch

or

switch = *value*

where

switch is a name formed according to standard Pascal rules for declaring identifiers. Note that a switch name applies only at the compiler level; therefore, you may declare an identifier of the same name in the program.

value is a whole number constant ranging from 0 to 255. This value is assigned to the *switch.* If the *value* and the equal sign (=) are omitted from the *switch* assignment, the default value true (0FFH) is assigned to the *switch.*

Example

This example of a SET control line sets the switch TEST to true (OFFH) and the switch ITERATION to 3. Declaring switches is optional.

```
$SET(TEST, ITERATION = 3)
```

10.3.19 SMALL

Specifies the memory addressing techniques of the program under compilation.

Syntax

```
1. SMALL {[(-CONST IN DATA-)]}
         { (-CONST IN CODE-)  }

2. SMALL ([subsystem-id][submodel]EXPORTS public-list[; EXPORTS public-list]...)

3. SMALL (subsystem-id[submodel] HAS module-list [{; HAS module-list     }]...)
                                                 [{; EXPORTS public-list }]
```

Abbreviation

SM

Default

LARGE (see 10.3.12)

Type

General

Description

The SMALL control directs the compiler to perform certain memory addressing optimizations that help reduce the amount of code produced. If you do not need to optimize your code size, or if you are not sure that your program meets the SMALL memory restrictions, simply use the default LARGE control. (For more advanced methods of program optimization, see Appendix I.)

This section discusses the SMALL control in its simplest form (see (1) above). The syntax in (2) and (3) applies only to the extended controls, which are discussed in Appendix I. Also see Appendix I for placement of these controls.

Modules compiled with the SMALL control have four sections, code, constant, data, and stack (see 11.2). When these modules are linked, similar sections from each module are combined to form two segments. A SMALL module has two segments: code and data.

In the default SMALL case (—CONST IN DATA—), the code sections from all the modules are allocated space within one segment, which is addressed relative to the CS register. All constant, data, and stack sections, as well as the heap, are combined in a second segment. This second segment is addressed relative to the DS register, with an identical copy in the SS register.

If (—CONST IN CODE—) is specified, the code and constant sections from all the modules are allocated space in one segment, which is addressed relative to the CS register. The data and stack sections are combined in a second segment. This segment is addressed relative to the DS register, with an identical copy in the SS register.

References to any location require only a 16-bit offset, using these segment addresses. In either case, (—CONST IN DATA—) or (—CONST IN CODE—), the maximum size of each segment is 64K. Since the two segments are fully defined by the time the program is loaded, the addresses in the CS, DS, and SS registers are never updated.

Dynamic variables are allocated on the heap. Note that SMALL (—CONST IN DATA—) uses a different heap mechanism than SMALL (—CONST IN CODE—), COMPACT, and LARGE. Since this SMALL heap is stored with the constant, data, and stack sections, dynamic variables are addressed with only 16-bit pointers. SMALL (—CONST IN CODE—) uses the same heap mechanism as COMPACT and LARGE; consequently, 32-bit pointers are required.

10.3.20 SUBTITLE

Puts a subtitle on each page of printed output.

Syntax

`SUBTITLE('`*subtitle*`')`

Abbreviation

ST

Default

no subtitle

Type

General

Description

The SUBTITLE control prints a subtitle on every page of printed output. To specify a subtitle, supply a sequence of printable ASCII characters (a string) for *subtitle*, enclosed within apostrophes.

The subtitle is placed on the subtitle line of each page of listed output, and is truncated on the right if necessary. The maximum length allowed for *subtitle* is 55 characters.

When a SUBTITLE control appears before the first non-control line in the source file, it puts the *subtitle* on the first page and on all subsequent pages until another SUBTITLE control appears. A subsequent SUBTITLE control causes a page eject, and the new subtitle is put on the next page and on all subsequent pages until another SUBTITLE control appears.

Examples

```
$SUBTITLE('INPUT ROUTINE')
.
.    (source lines)
.
$SUBTITLE('OUTPUT ROUTINE')
.
.
.
```

NOTE

If the NOLIST control is in effect, the *subtitle* is saved and appears again as a subtitle when the listing resumes.

10.3.21 SYMBOLSPACE

Specifies the amount of memory allocated for the static symbol area.

Syntax

```
SYMBOLSPACE(n)
```

Abbreviation

```
SS(n)
```

Default

```
SYMBOLSPACE(16)
```

Type

Primary

Description

The SYMBOLSPACE control specifies the amount of memory (in kilobytes) that is allocated for the compiler's static (or 'internal') symbol table. Values for *n* can range from 5 to 64, but a warning will be issued if the request exceeds the amount of available memory.

Note that this is not a measure of the total amount of memory to be used by the compiler. In systems with up to 192K (128K for the system plus a maximum of 64K for the static symbol table), increasing the value of *n* will decrease the amount of memory available for the dynamic symbol table. Adding more memory to the system will not increase the allocation for the static symbol table, but will be used to keep the dynamic symbol table from spilling to disk.

The static symbol area cannot spill to disk (only the dynamic table has this capability). If the static table runs out of memory, an error message will be generated and compilation will be aborted. The dictionary summary at the end of the listing will help determine how to adjust the SS(*n*) for your particular program if necessary.

10.3.22 TITLE

Prints a title on each page of printed output.

Syntax

```
TITLE('title')
```

Abbreviation

```
TT
```

Default

no title

Type

Primary

Description

The TITLE control prints a title on every page of printed output. To specify a title, supply a sequence of printable ASCII characters (a string) for *title*, enclosed within apostrophes.

The *title* is placed on the title line of each page of listed output, truncated on the right if necessary. The maximum length allowed for *title* is 55 characters.

Example

```
$TITLE('TEST PROGRAM 4')
```

10.3.23 TYPE/NOTYPE

Includes type records in the object module.

Syntax

```
TYPE
NOTYPE
```

Abbreviation

```
TY/NOTY
```

Default

```
TYPE
```

Type

Primary

Description

The TYPE control directs the compiler to include type records in the object module. These records describe attributes of symbols used in the source program, and they are used later for type checking by the linker. Type records provide a mechanism for enforcing type compatibility between separately compiled modules.

The NOTYPE control prevents the inclusion of type records in the object module.

Examples

```
$TYPE
```

This control line directs the compiler to include type records in the object module.

```
$NOTY
```

The NOTY control in this control line directs the compiler to *not* include type records in TEMP.OBJ.

10.3.24 XREF/NOXREF

Allows or prevents a cross-reference listing of source program identifiers.

Syntax

```
XREF
NOXREF
```

Abbreviation

```
XR/NOXR
```

Default

```
NOXREF
```

Type

Primary

Description

The XREF control directs the compiler to produce a cross-reference listing of all identifiers and labels in the source program. The compiler prints an entry for each Pascal constant, type, variable, parameter, procedure, function, or label that occurs in the source program, in alphabetical order. The listing is appended to the listing file created by the PRINT control (see PRINT/NOPRINT).

The NOXREF control prevents this cross-reference listing. The default setting is NOXREF.

Example

```
$XREF
```

The XREF control in this control line produces a cross-reference listing of all identifiers, and appends the listing to the output file specified by the PRINT control (or its default listing file, SOURCE.LST).

NOTE

Section 11.1.4 provides an example of a cross-reference listing.

CHAPTER 11
COMPILER OUTPUT

During the compilation process, the compiler produces a listing of the source program, and also an object module. The controls affecting the listing file and object file are described in Chapter 10. This chapter outlines the contents of both files.

11.1 Program Listing

Unless the NOPRINT control (described in 10.3.16) is active, the listing file is either the file specified in a PRINT control, or the default listing file (*source*.LST, where *source* is the name, without extension, of the source program file).

The listing file starts with a "sign-on" preface, then proceeds with the source listing, including any semantic error messages. If the XREF primary control is active, a symbol and identifier cross-reference listing is appended to the source listing. If the CODE control is active, the program listing also includes a listing of the approximate assembly code for the source code. The program listing always ends with a compilation summary.

Certain sections of the listing may not appear, depending on which controls are active. If NOPRINT is active, error messages are the only listing output produced. The error messages appear in the file specified in an ERRORPRINT control, or on the console if NOERRORPRINT or the default is in effect. You can use the NOLIST general control to isolate certain sections of the source code and not list them.

By default, the COND control is active, specifying that any source code skipped during conditional compilation will appear in the listing (without statement numbers or level numbers). A source listing produced while the NOCOND control is in effect does not provide a listing of the skipped source code. However, the controls that delimit the skipped source code are listed, indicating that code has been skipped.

Paging occurs automatically during the source, cross-reference, and assembly code listing, but you can force a page eject in the source listing by using the EJECT control. Each page holds 60 lines, with 120 characters per line.

Each page of the listing file has a numbered page header which identifies the compiler, the module and procedure being compiled, the date of the compilation, and (optionally) a title and subtitle you can provide with the TITLE and SUBTITLE controls.

In the listing, the procedure and module names in the page header are truncated to 24 characters, and the title and subtitle are truncated to 55 characters. If the procedure nesting exceeds 16 levels, the name that appears in the page header is the procedure at level 16.

11.1.1 Listing Preface

On the first page of the listing, below the header, the compiler prints a summary of the invocation line used to invoke the compiler, and the names of the source file and object file. If you specified the NOOBJECT control, no name is supplied for the object file.

Next to the heading "Controls Specified", the compiler lists the controls you specified in the invocation line. Figure 11-1 shows a sample listing preface.

```
Source File: PROG1.SRC
Object File: PROG1.OBJ
Controls Specified: <none>.
```

Figure 11-1. Sample Listing Preface

11.1.2 Source Listing

The section following the preface includes the source listing of the module being compiled, any errors detected during compilation, and an optional cross-reference listing of source program identifiers and symbols. Following the cross-reference listing is an optional listing of approximate assembly code. The source listing is described here, and the other descriptions follow.

The source listing contains a line-for-line copy of the source file, with some additional information. Figure 11-2 shows a sample partial source listing.

The leftmost column (under the heading STMT) contains the number of the first statement in the line. The compiler increments the statement number at every instance of a semicolon not contained within a parameter list, and every instance of a DO, THEN, ELSE, OTHERWISE, UNTIL, and an OF used in a CASE statement. The next column (under LINE) contains the ordinal position of the line in its source file. The next three columns (under NESTING) contain various measures of the nesting level (depth) of the first statement on the line. A source listing statement is not the same as a statement as defined in Chapter 7, since headings, declarations, and definitions have statement numbers in the source listing.

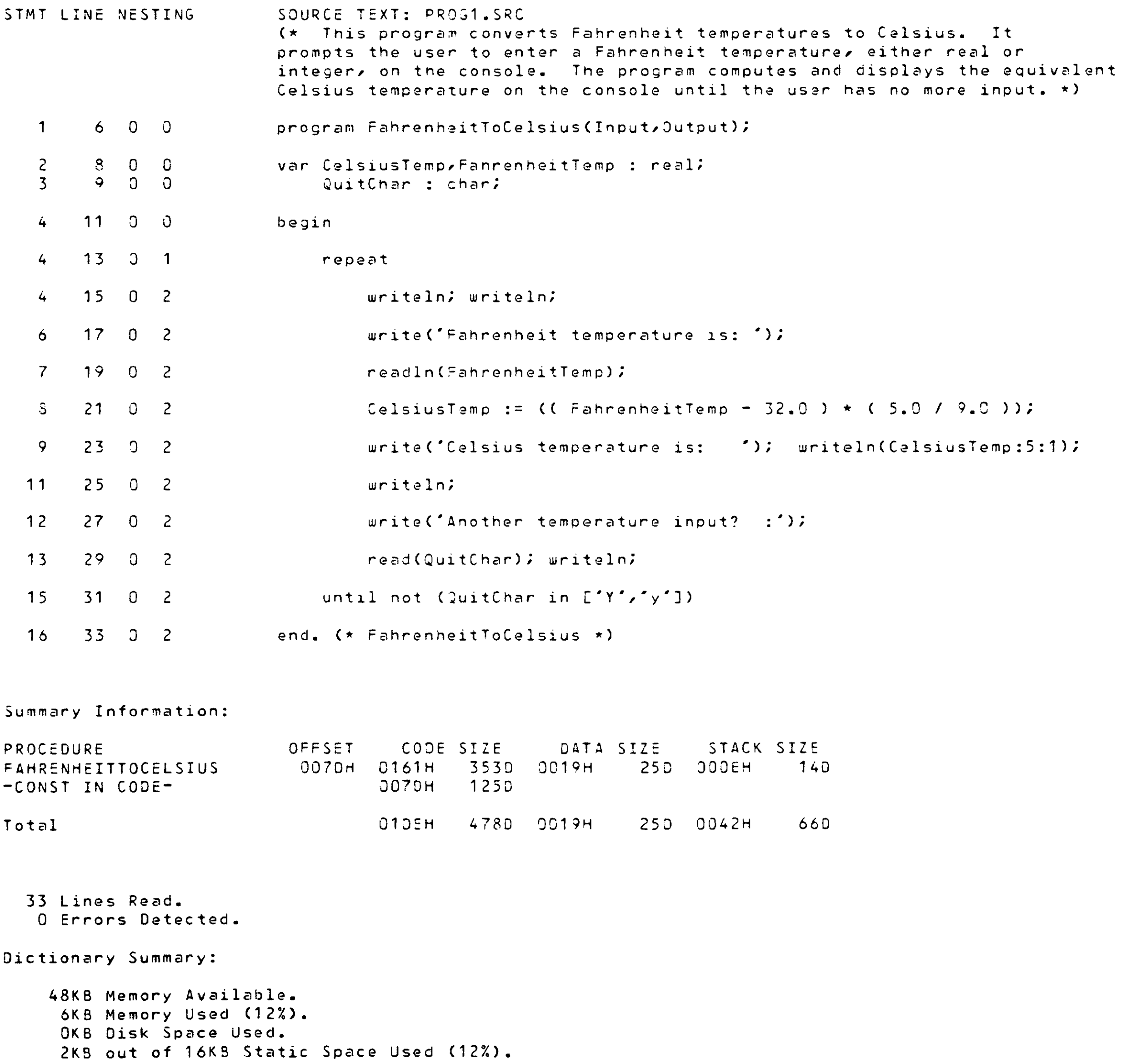
The first of these NESTING columns measures the procedure nesting depth of the first statement on the line. Statements and declarations at the module level are at level 0.

The second NESTING column measures the "block" nesting depth of the statement (BEGIN...END, REPEAT...UNTIL, and CASE...OF...END delimit Pascal "blocks" of statement nesting). This indicator always measures the depth for the first statement on the source line listed.

In the third NESTING column, any source lines included through use of the INCLUDE control (described in 10.3.8) are marked with "=*n*", where *n* is the nesting depth of the included line (*n* cannot be greater than 5).

If a source line is too long to fit on one listing line, it is continued on subsequent lines and preceded by a dash (—).

Except for the carriage return (CR), line feed (LF), and horizontal tab (HT), all nonprinting characters in the source are printed as *#nn#*, where *nn* is their hexadecimal value (see Appendix G). CR and LF are printed verbatim; HT sets tab stops every four columns.



```
STMT LINE NESTING        SOURCE TEXT: PROG1.SRC
                         (*  This program converts Fahrenheit temperatures to Celsius.  It
                         prompts the user to enter a Fahrenheit temperature, either real or
                         integer, on the console.  The program computes and displays the equivalent
                         Celsius temperature on the console until the user has no more input. *)

   1    6  0  0          program FahrenheitToCelsius(Input,Output);

   2    8  0  0          var CelsiusTemp,FanrenheitTemp : real;
   3    9  0  0              QuitChar : char;

   4   11  0  0          begin

   4   13  0  1              repeat

   4   15  0  2                  writeln; writeln;

   6   17  0  2                  write('Fahrenheit temperature is: ');

   7   19  0  2                  readln(FahrenheitTemp);

   8   21  0  2                  CelsiusTemp := (( FahrenheitTemp - 32.0 ) * ( 5.0 / 9.0 ));

   9   23  0  2                  write('Celsius temperature is:   ');  writeln(CelsiusTemp:5:1);

  11   25  0  2                  writeln;

  12   27  0  2                  write('Another temperature input?  :');

  13   29  0  2                  read(QuitChar); writeln;

  15   31  0  2              until not (QuitChar in ['Y','y'])

  16   33  0  2          end. (* FahrenheitToCelsius *)


Summary Information:

PROCEDURE                 OFFSET    CODE SIZE      DATA SIZE     STACK SIZE
FAHRENHEITTOCELSIUS        0070H   0161H    353D  0019H    25D  000EH    14D
-CONST IN CODE-                    0070H    125D

Total                              010EH    478D  0019H    25D  0042H    66D


  33 Lines Read.
   0 Errors Detected.

Dictionary Summary:

    48KB Memory Available.
     6KB Memory Used (12%).
     0KB Disk Space Used.
     2KB out of 16KB Static Space Used (12%).
```

Figure 11-2. Sample Partial Source Listing

11.1.3 Error Messages

If the compiler finds any errors during compilation, it reports the errors in the listing file, and also in the ERRORPRINT file if one was selected. If NOPRINT is active, the errors appear only in the ERRORPRINT file, or on the console if no ERRORPRINT file was selected.

With the exception of syntactic and lexical messages, each message appears on a line by itself in the following form:

```
****severity n[ IN stmt (file,line)]: message
```

The number *n* is a unique number for each *message*; all compiler error messages are described in Chapter 13. *Stmt* is the number of the statement, *file* is the name of the source module that contained the error, and *line* is the source line number. *Stmt*, *file*, and *line* do not appear in some error messages.

There are several levels of *severity*:

- EXTENSION messages, generated only when the NOEXTENSIONS control is active, show where your program deviates from standard Pascal. Compilation continues, since Pascal-86 supports these features.
- WARNING messages show areas of questionable quality in your programming style, but they do not stop compilation. Syntactic and lexical errors also fall into this category.
- ERROR messages show severe errors that prevent the generation of an object module.
- LIMIT EXCEEDED messages indicate that the compiler cannot generate an object module, and may not even continue with compilation.
- FATAL ERROR messages indicate anomalies in the compiler itself, or in the environment, that make it impossible to proceed with the compilation.

These severity levels are discussed in more detail in Chapter 13.

Syntactic and lexical error messages which fall in the WARNING category are reported and corrected as follows:

```
***WARNING, input: "error"
***was repaired to "repair"
```

The *error* is the sequence of tokens read from the source input, and the *repair* is the sequence accepted by the parser or scanner. An unprintable illegal character appears in hexadecimal notation enclosed by pound (#) signs (e.g., #03# for the CNTL-C key combination).

Scanner and parser error messages are always interleaved with the source listing, with each message appearing under the line or line segment that generated the error.

The following are examples of error messages as they would appear in the listing file:

If the source input line is:

```
IF X=0\IF Y=1 THEN X:=X+1;
```

then the message generated in the listing file would be:

```
 224  112  3  2         IF X=0\
*** WARNING, input: "IF X = 0 \ "
*** was repaired to "IF X = 0 "
 225  112  3  2               IF Y=1
```

```
*** WARNING, input: "IF Y = 1 "
***was repaired to "THEN IF Y = 1 "
 226   112  3  2                       THEN X:=X+1;
```

11.1.4 Symbol and Identifier Cross-Reference Listing

If you specify the XREF control, the compiler will generate a symbol and identifier cross-reference listing and append it to the source listing.

The compiler prints an entry for each constant, type, variable, parameter, procedure, function, and label that appears in the source program. They appear in alphabetical order by *name*, where *name* is the identifier or label number used in the source program.

The Pascal built-in procedures and functions that are implemented in-line or by calls to non-standard procedures are not listed. These include: ABS, CAUSEINTERRUPT, DISABLEINTERRUPTS, DISPOSE, ENABLEINTERRUPTS, EOF, EOLN, GET, GET8087ERRORS, INBYT, INWRD, LORD, MASK8087ERRORS, NEW, ODD, ORD, OUTBYT, OUTWRD, PACK, PAGE, PRED, PUT, READ, READLN, RESET, REWRITE, SETINTERRUPT, SQR, SQRT, SUCC, UNPACK, WRD, WRITE, WRITELN.

Record field names do not appear in this listing, and are only referred to by the record variable used to access the record field.

Each entry consists of the following information:

name offset length [?] *scope type kind* IN *nabor* AT *stmt*:
READ *readref*; WRITE *writeref*

where

name is the source identifier or label number for the entity. If this *name* is followed by an up-arrow (↑), the entity is a dynamic variable.

offset is the entity's hexadecimal offset (only supplied for PUBLIC and local entities of any *kind* except LABEL, PROCEDURE, FUNCTION, TYPE, or INTEGER CONSTANT). For CONSTANT, it is the offset of the constant from the start of the constant segment. For other *kinds*, it is the offset of the entity from the start of the data segment (if *nabor* is the name of the module) or stack frame (if *nabor* is the name of a procedure or function).

length is the number of bytes of storage occupied by the entity.

scope is PUBLIC, EXTERNAL, or nothing (to indicate a local entity).

type is the TYPE *type-defn* as it appears in the source program (user-defined type identifiers are used wherever applicable); however, record types are denoted only by RECORD. This field is omitted if the entity's *kind* is LABEL, PROCEDURE, or PROCEDURAL PARAMETER.

kind is LABEL, CONSTANT, TYPE, VARIABLE, PROCEDURE, FUNCTION, PARAMETER, VARIABLE PARAMETER, PROCEDURAL PARAMETER, FUNCTIONAL PARAMETER, MODULE, or PROGRAM PARAMETER.

nabor is the entity's "neighborhood": the name of the module, procedure, or function that contains the entity (includes the name of the subsystem containing the entity, if appropriate).

stmt is the number of the statement where the entity is defined.

readref
writeref are two lists of statement numbers: one for read references to the entity, and one for write references.

A question mark (?) will appear to the left of a symbol's attribute listing (see figure 11-3) if there are no references to that symbol in the compilation.

Figure 11-3 shows a sample cross-reference listing.

11.1.5 Listing of Approximate Assembly Code

If the CODE general control is active (and NOPRINT and NOOBJECT are not active), the compiler lists the approximate assembly language code that is equivalent to the object code produced. This listing occurs after the cross-reference listing and appears in six columns of information:

- Location counter (in hexadecimal notation)
- Resultant binary code (in hexadecimal notation)
- Label field
- Opcode mnemonic
- Symbolic arguments
- Comment field

```
                                        Cross-Reference Listing

Name              Offset  Length        Attributes and References
--------------    ------  ------        -------------------------

ADDNODE. . . .                          procedure in BUILDTREE at 15; read: 29 33.
BOOLEAN. . . .                 1        primitive type; read: 14.
BUILDTREE. . .                          procedure in TREETRAVERSAL at 13; read: 68.
CHAR . . . . .                 1        primitive type; read: 5 9.
DATAFILE . . .       12H       8        TEXT variable in TREETRAVERSAL at 12; write: 19 21 23 30 62 66; read: 34 67.
EXPRESSIONTREE       1AH      63        TREE variable in TREETRAVERSAL at 11; write: 18 19 21; read: 17 20 22 37 38 40 41
                                        42 45 47 48 49 50 51 53 55 56 57 58 59 61.
FALSE. . . . .                 1        predefined BOOLEAN constant; read: 26.
FINDROOT . . .     FFFDH       1        BOOLEAN variable in BUILDTREE at 14; write: 26 32; read: 34.
INFIX. . . . .                          procedure in TREETRAVERSAL at 36; read: 40 42 71.
INTEGER. . . .                 2        primitive type; read: 10.
MAXNUMNODES. .                 2        INTEGER constant in TREETRAVERSAL at 2; read: 4.
NODE . . . . .                 3        record type in TREETRAVERSAL at 5; read: 8.
NODECHARACTER.       59H       1        CHAR variable in TREETRAVERSAL at 9; write: 30 66; read: 16 18.
NODEINDEX. . .        4H       1        SUBSCR parameter in INFIX at 36; read: 37.
NODEINDEX. . .        4H       1        SUBSCR parameter in POSTFIX at 54; read: 55.
NODEINDEX. . .        4H       1        SUBSCR parameter in PREFIX at 46; read: 47.
NODEINDEX. . .       10H       2        INTEGER variable in TREETRAVERSAL at 10; write: 30 66; read: 16 17 31.
ONE. . . . . .                 2        INTEGER constant in TREETRAVERSAL at 3; read: 71 74 77.
OUTPUT . . . .        0H       8        predefined TEXT variable; read: 16 20 22 24 27 28 35 39 41 43 45 49 53 59 61 63
                                        64 65 69 70 72 73 75 76 78 79 81 82.
POSTFIX. . . .                          procedure in TREETRAVERSAL at 54; read: 57 58 77.
PREFIX . . . .                          procedure in TREETRAVERSAL at 46; read: 50 51 74.
SUBSCR . . . .                 1        0 .. 20 type in TREETRAVERSAL at 4; read: 6 7 8 36 46 54.
TEXT . . . . .                 8        primitive type; read: 12.
TREE . . . . .                63        array[ SUBSCR ] of NODE type in TREETRAVERSAL at 8; read: 11.
TRUE . . . . .                 1        predefined BOOLEAN constant; read: 32.
```

Figure 11-3. Sample Cross-Reference Listing

Not all of these columns will appear on every line of the approximate assembly code listing. Compiler-generated labels are preceded by (?). The code generated from each source statement will be headed by a comment line bearing the statement number of that source statement. Figure 11-4 shows a sample listing of approximate assembly code.

11.1.6 Compilation Summary

The compilation summary appears at the end of the listing, and provides the following information for each procedure, function, and module:

- The offset, in hexadecimal, of the entity's entry point in the code section
- The size, in hexadecimal and decimal, of the code section
- The size, in hexadecimal and decimal, of the constant section
- The size, in hexadecimal and decimal, of the data section
- The size, in hexadecimal and decimal, of the stack section

The summary also lists the percentage of free memory used by the compilations, the number of source lines read and included with the INCLUDE control, and the number of errors detected. Figure 11-5 shows a sample compilation summary.

The summary also lists the number of source lines read and included with the INCLUDE control, and the numbers of errors detected. The dictionary summary provides information about how memory is utilized. Memory Available represents how much system memory is available to the dynamic symbol table. Memory Used states how much of that memory is actually used. Disk Space Used indicates the amount of disk storage used for the symbol table. Static Space Used gives how much memory is used by internal tables against how much is allocated. This last number is determined by the SYMBOLSPACE control. Figure 11-5 shows a sample compilation summary.

11.2 Object Module

The result of a successful compilation is a file containing a relocatable object module. You link this file with the Pascal run-time libraries and other relocatable files (as described in Chapter 12) to produce a single file that is executable.

From the source file and any included files, the compiler produces one object file that contains one object module as the result of the compilation. This object module contains sections as described below, which are unique to each module.

The object module has the following sections:

- Code Section
- Constant Section
- Data Section
- Stack Section

These sections can be combined in various ways into "memory segments" for execution, depending on the control used when compiling the program: SMALL (10.3.18), COMPACT (10.3.3), or LARGE (10.3.12).

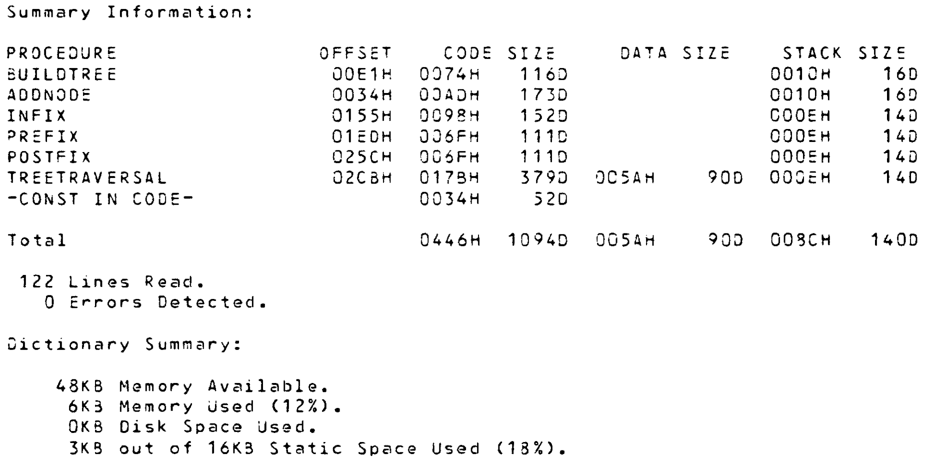
```
                                        Assembly Listing of Generated Object Code

                        ; STATEMENT # 1

                        ; STATEMENT # 11

                        RESETCOUNT
                                PROC NEAR
0000  55                        PUSH    BP
0001  8BEC                      MOV     BP,SP
0003  55                        PUSH    BP
0004  83EC02                    SUB     SP,2H

                        ; STATEMENT # 15

0007  C746FC0C30                MOV     COUNT[BP],300CH

                        ; STATEMENT # 16

000C  8A46FC                    MOV     AL,COUNT[BP]
000F  E6D0                      OUT     0D0H

                        ; STATEMENT # 17

0011  8A46FD                    MOV     AL,COUNT[BP+1H]
0014  E6D0                      OUT     0D0H
0016  8BE5                      MOV     SP,BP
0018  5D                        POP     BP
0019  C3                        RET
                        RESETCOUNT
                                ENDP

                        ; STATEMENT # 18

                        INITIALIZECHIP
                                PROC NEAR
001A  55                        PUSH    BP
001B  8BEC                      MOV     BP,SP
001D  55                        PUSH    BP
001E  83EC02                    SUB     SP,2H

                        ; STATEMENT # 20

0021  B030                      MOV     AL,30H
0023  E6D6                      OUT     0D6H

                        ; STATEMENT # 21

0025  E8D8FF                    CALL    RESETCOUNT

                        ; STATEMENT # 22

0028  E4C2                      IN      0C2H
002A  8846FD                    MOV     IMASK[BP],AL

                        ; STATEMENT # 23

002D  B004                      MOV     AL,4H
002F  F6D0                      NOT     AL
0031  2246FD                    AND     AL,IMASK[BP]
0034  E6C2                      OUT     0C2H

                        ; STATEMENT # 24

0036  8BE5                      MOV     SP,BP
0038  5D                        POP     BP
0039  C3                        RET
                        INITIALIZECHIP
                                ENDP

                        ; STATEMENT # 25
```

Figure 11-4. Sample Listing of Approximate Assembly Code

In addition, there may be special records generated through your use of the DEBUG and/or TYPE controls, and public and external entities. These records are discussed after the following descriptions of object module sections.

11.2.1 Code Section

This section contains the object code generated by the source program. If the object module is a main module, the code section also contains a "main module prologue" generated by the compiler. This "prologue" precedes the code compiled from the



```
Summary Information:

PROCEDURE                    OFFSET   CODE SIZE     DATA SIZE    STACK SIZE
BUILDTREE                     00E1H  0074H   116D                0010H    16D
ADDNODE                       0034H  00ADH   173D                0010H    16D
INFIX                         0155H  0098H   152D                000EH    14D
PREFIX                        01EDH  006FH   111D                000EH    14D
POSTFIX                       025CH  006FH   111D                000EH    14D
TREETRAVERSAL                 02CBH  017BH   379D  005AH    90D  000EH    14D
-CONST IN CODE-                      0034H    52D

Total                                0446H  1094D  005AH    90D  008CH   140D

 122 Lines Read.
   0 Errors Detected.

Dictionary Summary:

    48KB Memory Available.
     6KB Memory Used (12%).
     0KB Disk Space Used.
     3KB out of 16KB Static Space Used (18%).
```

Figure 11-5. Sample Compilation Summary

source program, and contains the code to set up the CPU for program execution by initializing various registers and enabling interrupts. Information about logical files used in the module is stored in the code section.

If either the SMALL or COMPACT control is used, the code sections from all modules are combined and allocated space in one logical segment. (Logical segments are described in 12.1.)

If the LARGE control is used (the default), the code section for each module forms a complete logical segment that is unique for each module.

11.2.2 Constant Section

The constant section contains all memory-resident constants, both literal constants and those defined with the keyword CONST.

If either the SMALL or COMPACT (—CONST IN CODE—) control is used, the constant sections from all modules are combined in the same logical segment as the code sections. If (—CONST IN DATA—) is specified, these constant sections are combined in the same logical segment as the data sections (11.2.3). (Logical segments are described in 12.1.)

If the LARGE (—CONST IN CODE—) control is used (the default), the constant section for each module is combined with the code section to form a complete logical segment that is unique for each module. If (—CONST IN DATA—) is specified, the constant section is combined with the data section to form a complete logical segment that is unique for each module.

11.2.3 Data Section

The data section contains all global variables at level 0 (above the level of procedures). All variables at level 1 or deeper (below the level of procedures) are stored in the stack section.

If either the SMALL or COMPACT control is used, the data sections from all modules are combined and allocated space in one logical segment. (Logical segments are described in 12.1.)

If the LARGE control is used (the default), the data section for each module forms one or more complete logical segments that are unique for each module.

11.2.4 Stack Section

The stack section contains all parameters, all variables at level 1 or deeper (below the level of procedures), and is used to store temporary information. The exact size of the stack required by each module is estimated by the compiler as the sum of the STACK sections for each procedure in the module. You can override this computation and explicitly state the stack requirement during the location process.

If the SMALL control is used, the stack sections from all modules are combined with the data sections and allocated space in one logical segment. (Logical segments are described in 12.1.)

If the COMPACT or LARGE control is used, the stack sections from all modules are combined and allocated space in one logical segment.

NOTE

> Since Pascal's procedures are by definition reentrant, you must be careful to allocate a stack section large enough to accommodate all possible storage required by multiple incarnations of such procedures. The stack size can be explicitly specified during the location process, and you can find stack size information in the compilation summary.

If a run-time stack overflow exception occurs, you may not have allocated enough stack space for using sets. The maximum size for a set is $(N/8) + 6$ bytes where n is the number of set members. The number of sets on the stack is a function of the complexity of the set expressions in your program. To increase stack size, use the SEGSIZE control of LINK86. For example, if you are using sets of type 1..1000, $N = 1000$ and the maximum size is $(1000/8) + 6$ or 134 bytes. Since two sets are usually maintained on the stack to implement set operations, the total space you should add is 268 bytes.

11.2.5 Additional Information

The compiler generates special records to hold *type* information if you specify the TYPE control. These type records are used by the linker and/or various debuggers to check for consistency between separately compiled modules, and you should make use of them if you want type checking to occur.

The compiler generates special *debug* records if you specify the DEBUG control. These records define local symbols and line numbers, which may be used later for symbolic debugging.

For each procedure and data structure appearing in a PUBLIC section of the module being compiled, the compiler generates PUBLIC name definition records. If the module being compiled is a main program module (see 2.2), a PUBLIC name definition is generated for the main program entry point. For each procedure and data structure that appears in another module but is referenced by the module currently being compiled, the compiler generates an external name definition record. These external name definition records are also generated for library routines referred to by the module currently being compiled. These records provide information to the linker, as described in Chapter 12.

CHAPTER 12 LINKING, LOCATING, AND EXECUTING PROGRAMS

12.1 Introduction

An important feature of Pascal-86 is the ability to compile separate object modules that are parts of a whole program. The iAPX 86 utilities provide a way to link these modules with modules from run-time support libraries (and modules translated from other languages) to form the whole program.

The result of a single compilation is an object module, but it does not have to contain a complete program. In fact, to perform file input/output, set and packed data manipulation, real arithmetic, and dynamic memory management (NEW and DISPOSE), your object module has to refer to other modules included in *run-time support libraries*, which are listed in 12.2.2.

In many cases, you will want to write portions of a Pascal program and execute and debug them separately. You might also want to code portions of a program in another language, translate those portions separately, and link all portions to produce the final program.

To execute Pascal programs, you have to *link* the object modules that are needed, and *locate* them in memory (bind them to memory addresses).

The operating system provides 8086-based utilities that allow you to link modules together, locate them in memory, and load them for execution. The linker (LINK86) links object modules and outputs a module to be located before loading, or located by the loader. The locater (LOC86) assigns absolute addresses to modules to locate them in actual memory. The loader loads and executes the final program. In addition, the LIB86 utility is provided to create and maintain your own library files of compiled (or translated) object modules.

The 8086 resident linker and locater are described in detail in the *iAPX 86, 88 Family Utilities User's Guide*, which also gives an overview of 8086/8088 memory addressing techniques; definitions of segments, classes, and groups; discussions of segment, class, and group combining; and descriptions of how the locater binds segments to addresses. The *iAPX 86, 88 Family Utilities User's Guide* also describes the mechanics of loading and executing, the maintenance of program libraries using the 8086-resident library utility (LIB86), the object code-to-hexadecimal conversion utility (OH86), and intermodule cross-reference (CREF86).

The following sections briefly describe the process of linking and locating Pascal-86 programs and using run-time libraries.

12.2 Linking Object Modules

You use the linker to link groups of logical segments in the order you choose, resolve all references to modules linked together, and prepare the final linked program for the locating operation.

The 8086-based linker (LINK86) will link separately compiled Pascal modules with other Pascal-86 modules, modules translated from other high-level languages like PL/M-86, and modules translated by ASM86 and the 8089 Macro Assembler.

To satisfy a module that contains external references to other modules, the linker uses the information compiled from the module's interface specification to find another module that contains a public symbol to match the external reference.

The linker produces a single output module. It combines logical segments with the same name, combines groups with the same name. The linker also selects modules from specified libraries to resolve external references, and optionally purges public symbol, local symbol, line number, and comment definitions from the output module. Throughout the process, the linker generates a link map and error messages for abnormal conditions.

The linker combines logical segments in the order in which they are encountered in the input modules, and on the complete logical segment name (the logical segment name and class name). The output module consists of one or more logical segments in the order in which unique segment names were encountered in the input modules. When a non-unique segment name (a name previously read) is encountered, the linker combines the logical segment with the segment previously read. The only way that you can change this sequence is to change the names of the logical segments, or change the order in which modules are specified in the linker's command line.

12.2.1 Use of Libraries

Libraries are files containing object modules that are created and maintained by the library utility, LIB86. These object modules contain public procedures that are referenced by many programs, i.e., they are common to most programs. You use them to build your programs by referring to them as external procedures in your programs, and linking them to your programs.

The linker treats library files in a special manner. When you specify input modules to the linker, the linker keeps track of all external references. Then, when you specify a library file as input to the linker, the linker searches the library for modules that satisfy these unresolved external references. This means that libraries should be specified to the linker *after* the input modules that contain external references. If a module in a library has an external reference, the linker searches the library again to try to satisfy the reference. The process continues until all external references are satisfied, or until the linker cannot find any more public symbols to satisfy an external reference

The library utility is described in detail in the *iAPX 86, 88 Family Utilities User's Guide.*

12.2.2 Run-Time Support Libraries

Intel supplies libraries to provide run-time support for Pascal-86 modules that perform file input/output, set and packed data manipulation, arithmetic functions, real arithmetic, and dynamic memory (heap) management. The run-time support is divided into separate libraries so that only the code required for your application is linked in. You do not have to maintain these libraries using LIB86, since they are supplied as libraries.

For example, you would link in the library 8087.LIB if you are using the 8087 Numeric Data Processor for real arithmetic. If you are using the 8087 emulator, you would link in the library E8087.LIB and the module E8087 instead. If you are not performing any real arithmetic, you would link in the library 87NULL.LIB instead.

The run-time libraries supplied are as follows. They should be linked in the order listed.

- P86RN0.LIB and P86RN1.LIB are both required for all run-time support.
- P86RN2.LIB and P86RN3.LIB are the required logical record system libraries. If you intend to provide the interface (logical record interface) to your own record system, see K.3.7. If you are not providing run-time support and are not using any predefined Pascal input/output, or dynamic memory functions, you must link in RTNULL.LIB to resolve external references.
- CEL87.LIB is required to support the following built-in functions: EXP, LN, SIN, COS, TAN, ARCSIN, ARCCOS, ARCTAN, TRUNC, ROUND, LTRUNC, and LROUND.
- EH87.LIB is required to implement IEEE standard math features that the 8087 does not support. These include the normalized mode of arithmetic and nontrapping NaN support.
- 8087.LIB is required to support real arithmetic with the 8087 Numeric Data Processor. If you are using the 8087 emulator, use E8087.LIB and the module E8087 instead. If you are not performing real arithmetic, you must link in 87NULL.LIB to resolve external references.
- LARGE.LIB is required to execute Pascal-86 programs in the Series III environment; it supplies the UDI interface. If your program will run in an environment other than the Series III, you need to link in the UDI interface for your operating system. You do not need LARGE.LIB if you linked in RTNULL.LIB (for no run-time support), unless you supplied your own run-time support libraries that rely on the Series III operating system.

Reentrancy

All of the Pascal-86 run-time system is reentrant except for P86RN0.LIB. A copy of this library must be linked into each task, though all of the other libraries can be shared between tasks. Each task must be initiated by one "main program" or its equivalent (see Appendix J) which will initialize the task's copy of the shared routines. For details on how to use the run-time libraries in a multitasking environment, see the *Run-Time Support Manual for iAPX 86,88 Applications*, Order Number 121776.

12.2.3 Position-Independent and Load-Time Locatable Modules

The Pascal-86 compiler attempts to produce *position-independent code* (*PIC modules*) for non-main modules that do *not* use any file input/output, packed data structures, NEW and DISPOSE to allocate and de-allocate heap space, real arithmetic, procedural parameters, or set operators. Position-independent modules can be located by the loader, then executed.

Any modules that use any of these features, and all main modules, have to employ 4-byte long pointers that address other modules. Hence, they are not position-independent, since their pointers need to know the base address of the memory segment that holds the other object module segments.

In certain stages of program development, you may want to produce modules that can be loaded anywhere in memory by the loader. Use the linker with the BIND control, as described in the *iAPX 86, 88 Family Utilities User's Guide*, to produce a program that can be located at load time by the loader.

A position-independent module cannot refer to another segment base, but *load-time locatable* modules can refer to segment bases in order to access other segments. A load-time locatable module is called an *LTL module*. You can create LTL modules using the BIND control with LINK86 to produce an output module that can be located by the loader. In this case, the loader decides where to load the linked LTL modules, resolves all references to segment bases, loads the program, initializes all segment registers, and executes the program in the iAPX 86 environment.

Consult the *iAPX 86, 88 Family Utilities User's Guide* for details on position-independent (PIC) and load-time locatable (LTL) modules.

12.2.4 Sample Link Operations

The following examples show how to execute Pascal-86 programs in different environments. Note that the operating system prompt (and loader, if applicable) are not included in these invocations (see your specific operating-system appendix).

1. To execute a Pascal-86 program in a full-featured operating system environment, you should link in all of the Pascal-86 run-time support libraries. If your application also requires support for floating-point arithmetic, you must link in the appropriate numerics libraries. For example, using the 8087 emulator, the link sequence would be:

```
LINK86 MYPROG.OBJ, P86RN0.LIB, P86RN1.LIB, P86RN2.LIB, & <cr>
** P86RN3.LIB, CEL87.LIB, E8087.LIB, E8087, & <cr>
** system-lib to MYPROG.86 BIND <cr>
```

   and using the 8087 chip, it would be:

```
LINK86 MYPROG.OBJ, P86RN0.LIB, P86RN1.LIB, P86RN2.LIB, & <cr>
** P86RN3.LIB, CEL87.LIB, 8087.LIB, & <cr>
** system-lib to MYPROG.86 BIND <cr>
```

   where

   system-lib is any interface library that may be required by your operating system (e.g., LARGE.LIB for the Series III).

   Both of these configurations fully support all of the features of Pascal-86. By using the BIND option with LINK86, the output file is ready to be executed, assuming that your operating system has an LTL loader.

   If your programs require floating-point support and you want to implement such IEEE standard math features as normalized arithmetic and non-trapping NaN support, you need to link in the 8087 exception handler EH87.LIB. For example, using the 8087 chip, your link command would be:

```
LINK86 MYPROG.OBJ, P86RN0.LIB, P86RN1.LIB, P86RN2.LIB, & <cr>
** P86RN3.LIB, CEL87.LIB, EH87.LIB, 8087.LIB, & <cr>
** system-lib to MYPROG.86 BIND <cr>
```

2. To execute your Pascal-86 program in a bare machine (or minimal operating system) environment, you only need to link in the run-time libraries P86RN0.LIB and P86RN1.LIB. If your program requires numerics support and you are using the 8087 chip, your link command would be:

```
LINK86 MYPROG.OBJ, P86RN0.LIB, P86RN1.LIB, RTNULL.LIB, & <cr>
** 8087.LIB TO MYPROG.LNK <cr>
```

   In this example, sets and 32-bit arithmetic operations are fully supported. Pascal input/output (e.g. READ, WRITE) and memory management (NEW and DISPOSE) are not supported; if used, LINK86.86 will generate an UNRESOLVED EXTERNALS warning. To support NEW and DISPOSE in this environment, see the *Run-Time Support Manual for iAPX 86,88 Applications.*

If you link in numerics support and an 8087 exception occurs, RTNULL.LIB will simply execute an HLT instruction. Since there are no external references between RTNULL.LIB and EH87.LIB, the exception handler will never be called. Consequently, it should not be included in the link sequence.

Note that the BIND option was not used, since in this environment your programs would probably be located and burned into ROM, or loaded with a simple absolute loader.

12.3 Locating Programs

The linker produces a single output module. This module can either be located by the 8086-based locater (LOC86) or, if the BIND control was used when linking the program, it can be loaded by the operating system's loader, then executed. The program must be located with LOC86 if it will be burned into ROM or loaded through ICE-86A.

The 8086-based locater (LOC86) binds locatable logical segments to absolute addresses. The locater creates an absolute output module from a single input module, generates a memory map that summarizes the results of address binding, generates a symbol table that shows the addresses of certain symbols, detects any errors that arise in the locating process, and filters locating information and compiler-generated debugging information.

The input module to the locater is usually the output module from the linker, but it could also be the direct result of a single compilation. The absolute output module is the program you can load and execute.

The locater assigns a physical address to each logical segment. You can direct the locater to place logical segments, classes of logical segments, or groups of logical segments in specific memory locations. The locater resolves logical addresses to physical addresses so that the same segment register can refer to more than one logical segment, if those logical segments are combined in a group. For an overview of 8086 memory addressing techniques, see the *iAPX 86, 88 Utilities User's Guide.*

You can collect logical segments into groups that fit into 64K. You can also use a class name to refer to logical segments that have the same attribute (e.g., all CODE segments, where the class name is CODE). The segment, group, and class names assigned by the Pascal-86 compiler depend on the segmentation model used, as described in Appendix J.

You can specify any or all addresses for the various logical segments, or specify none at all. The locater applies a default ordering and addressing assignment algorithm to those logical segments not mentioned in the LOC86 invocation line. The logical segments are ordered and assigned addresses in this sequence:

- The classes and logical segments mentioned in the ORDER control
- The classes and logical segments mentioned in the ADDRESSES control
- The logical segments that already have absolute addresses
- The class name of each logical segment
- The overlay name of each logical segment
- The parent group of an LTL module (if any)
- The order of logical segments in the input module

The locating process is described in detail in the *iAPX 86, 88 Family Utilities User's Guide.*

12.3.1 Locating the 8087 Emulator

The 8087 emulator object code is divided into classes: AQMCODE (the read-only portion) and AQMDATA (the read-write portion), and STACK. If you are using the 8087 emulator, you may want to locate these classes separately using LOC86, in order to locate the read-only portion (AQMCODE) in ROM and the read-write portion (AQMDATA) in RAM. Do this with the CLASSES subcontrol of the ADDRESSES control for LOC86.

If you are locating the 8087 emulator separately from your program, the interrupt vectors that your program uses must be initialized to point to the proper routines in the emulator. The INITFP routine, called as part of the main program prologue, will initialize these vectors if it knows the E8087 addresses. If the main program is located at the same time as the emulator, these addresses are readily available. Otherwise, use the PUBLICSONLY input control of LINK86 to obtain the located addresses of the emulator.

The emulator assumes that there are 180 bytes available on the stack for its use, which can be allocated using the SEGSIZE control of LOC86. The emulator also reserves interrupts 20 through 31.

12.4 Preconnecting Files

You can assign physical file names to file variables that are used as program parameters by using a mechanism known as *file preconnection.* A file variable is a program parameter if its identifier appears in a PROGRAM statement. You preconnect files at run time on the command line used to invoke your program.

Pascal-86 offers two ways to associate file variables with physical files: (1) a second parameter to REWRITE and RESET (an Intel extension to Pascal), used in a program to explicitly state the physical file name, and (2) the preconnection mechanism (standard to Pascal) used when you execute the compiled program. The first way takes precedence: if you supply a second parameter to REWRITE or RESET, any preconnection for that file variable is ignored. The second parameter to REWRITE and RESET is described in 8.7.1 and 8.7.2.

Default file names are provided for all file variables. If the file variable is a program parameter, you can override the default name by using the preconnection mechanism. You cannot preconnect physical files to other file variables, even if the other file variables are not specified in a second parameter to REWRITE or RESET. Temporary files are created by the host system for such file variables, and the temporary files are deleted when the program terminates.

To use the file preconnection mechanism, specify the file names on the program's invocation line.

The file preconnection format takes the form:

(*identifier* = *pathname*[, ...])

where

identifier is the file variable used as a program parameter.

pathname is a legal pathname for a physical file.

You do not have to specify or preconnect the physical devices :CO: and :CI: for the standard files INPUT and OUTPUT.

CHAPTER 13
COMPILE-TIME ERRORS AND WARNINGS

This chapter describes all of the error and warning messages produced by the compiler. Compiler messages are coded by number and are listed in this chapter in numeric order by their code numbers so that you can easily look up any message you received.

In addition to compile-time and run-time error messages (see Chapter 14), you may encounter other error messages during program development—errors during the linkage and location processes, and other operating system error conditions. Consult the *iAPX 86,88 Utilities User's Guide* for information on linkage and location errors and errors that occur while using the utilities.

13.1 General Format

Most of the errors and warnings reported by the compiler appear in the listing file governed by the PRINT control, *and* on the device or in the file governed by the ERRORPRINT control (which sends them to the console by default). If you use the NOPRINT control, the messages appear at the console only, or at a file specified in an ERRORPRINT control. If you specify NOERRORPRINT, errors and warnings are reported in the file governed by the PRINT control, or at the console if you specified NOPRINT.

Errors in the invocation line and in the first set of primary control lines are the only errors that are not governed by the PRINT or ERRORPRINT controls. These errors appear at the console immediately, if they occur.

The following two sections describe the invocation line errors and compile-time errors that are detected by the compiler. All errors and warnings reported by the compiler take the following form:

```
*** severity n IN stmt (file, LINE line): text
```

where

severity is either EXTENSION, WARNING, ERROR, LIMIT EXCEEDED, or FATAL ERROR.

n is the error number described below.

stmt is the statement number of the Pascal source statement where the error occurred.

file is the name of a source file.

line is the ordinal position of the source line in the source file where the erring statement resides.

text is the text of the message, as described below.

The error number *n* is a four-digit number where each digit is meaningful. The leftmost digit (thousandths' digit) indicates the severity of the message. Table 13-1 shows the corresponding severity for each leftmost digit of *n*.

The second leftmost digit (hundreds' digit) of the error number *n* identifies the phase of compilation in which the error occurred. Table 13-2 shows the corresponding phases.

Table 13-1. Severity Levels of Compiler Errors

Leftmost Digit	Severity	Meaning
1 or 2	EXTENSION	The compiler detects a violation of the ANSI/ IEEE770X3.97-1983. These messages occur only if the NOEXTENSIONS control is used.
3 or 4	WARNING	The compiler detects a bad situation, but the problem will not affect the validity of the generated code.
5 or 6	ERROR	The compiler detects a definite violation that invalidates the generated code.
7	LIMIT	A capacity limit of the compiler has been exceeded. Although compilation continues, the object code is not valid.
8	LIMIT	A capacity limit of the compiler has been exceeded, and compilation is aborted.
9	FATAL	The compiler detects an unexpected condition in the supporting environment. Compilation is aborted.

Table 13-2. Error Numbers Corresponding to Compilation Phases

Second Leftmost Digit	Phase of Compilation
0	Invocation Line and First Set of Primary Controls
1 or 2	Scanner
3	Parser
4	(not used)
5	Semanticist
6	Cross-Referencer
7	(not used)
8	Code Generator
9	Object Module Generator

13.2 Invocation Line and Primary Control Errors

The following error messages are also listed in numeric order in section 13.3, but are repeated here for quick referencing, since they will most likely occur immediately after you invoke the compiler.

FATAL ERROR 9000: I/O error on *filename*: *system information*

The compiler cannot find or open *filename*. Compilation is aborted and control is returned to the operating system. Check to see if the file exists.

FATAL ERROR 9001: Unable to open INCLUDE file
filename: *system information*

The INCLUDE file is not present, or is already open. Combination is aborted, and control is returned to the operating system.

FATAL ERROR 9002: I/O error on *filename*: *system information*

The compiler cannot find or open *filename*. Compilation is aborted and control is returned to the operating system. Check to see if the file exists.

FATAL ERROR 9005: Input file missing or syntax error in invocation line.

FATAL ERRORS 9006 to 9017: I/O error on compiler work file: *system information*

The compiler cannot open one or more of the eight files for some reason. Check to see if the files are assigned to another disk drive besides the designated work file drive, or if the proper disk is not in the designated work file drive. Use the WORK command to change the designated work file drive. If the error persists, contact your supplier. Compilation is aborted, and control is returned to the operating system.

FATAL ERROR 90*xx*: Compiler error in root.

Unexpected condition; contact your supplier. Compilation is aborted. (*xx* represents a particular number useful only to your supplier.)

FATAL ERROR 9201: Unknown control, *control*, in invocation group.

The "invocation group" includes the invocation line and the initial set of primary control lines in the source file. Compilation is aborted.

13.3 Compile-Time Errors and Warnings

FATAL ERROR messages and LIMIT messages over 8000 indicate that compilation was aborted. In all other cases, compilation will continue; however, ERROR messages indicate that an object-module was not generated. In ERROR messages, the compiler performs a corrective action that will enable it to continue compiling and looking for errors. One common action is to "neutralize" a type specification, so that subsequent references to the incorrect type do not cause more type incompatibility errors or undefined symbol errors. Objects whose type have been "neutralized" are called *neutral objects.*

When an EXTENSION or WARNING message occurs, the generated object code is valid as object code, but it might not be valid for the intended use of your program. When an ERROR or LIMIT message occurs, compilation may continue, but the object code is not generated. FATAL ERROR messages indicate that the compilation is aborted. In any case, if compilation is aborted, control is returned to the operating system.

EXTENSION 1201: Non-decimal integer constant, *const*.

The constant expression is not a decimal integer. Octal, binary, and hexadecimal integers are not permitted in standard Pascal. Compilation continues.

EXTENSION 1204: Non-standard compiler control.

This message will occur only if the compiler encounters a general control line. The initial set of primary control lines will not cause this message. Compiler controls are not part of standard Pascal. Compilation continues.

EXTENSION 1206: Non-standard underscore in identifier *identifier*.

Standard Pascal does not allow underscores in identifiers.

EXTENSION 1232: Non-standard LONGINT constant, *string*.

Standard Pascal requires that integer constants be in the range −32767 to +32767.

EXTENSION 1500: Non-standard interface specification.

Intel's Pascal-86 allows separate compilation units for a single program, each of which starts with an interface specification. Standard Pascal only supports a single compilation unit, which must start with a program heading.

EXTENSION 1502: Label too large.

To conform to the Pascal standard, labels must be in the range 0 to 9999. Intel's Pascal-86 does not have this restriction.

EXTENSION 1504: Non-standard concatenation of string constants.

Intel's Pascal-86 allows concatenation of string constants. Standard Pascal does not support this feature.

EXTENSION 1508: Case constants in variant record do not map onto range of tag type.

Standard Pascal requires that the set of case constant values in a variant record must equal the set of values in the tag type. Intel's Pascal-86 does not have this restriction.

EXTENSION 1514: Non-standard signature on forwarded definition.

Intel's Pascal-86 allows a parameter list (and result type) specification to appear in a procedure or function declaration that has been previously declared to be FORWARD. Standard Pascal does not support this feature.

EXTENSION 1516: Use of non-standard predefined symbol, *symbol*.

The *symbol* is a predefined constant, type, procedure, or function offered only in Pascal-86, not in standard Pascal. The symbols defined in Pascal-86 (that would cause this message) are TAN, ARCSIN, ARCCOS, INBYT, OUTBYT, INWRD, OUTWRD, GET8087ERRORS, MASK8087ERRORS, AT87ERRORS, AT87EXCEPTIONS, AT87PRCN, AT87UNDR, AT87OVER, AT87ZDIV, AT87DENR, AT87NVLD, AT87MASK, AT87RSVD, CR, LF, ENABLEINTERRUPTS, DISABLEINTERRUPTS, CAUSEINTERRUPT, SETINTERRUPT, LONGINT, MAXLONGINT, LTRUNC, LROUND, LORD, WORD, WRD, MAXWORD, TEMPREAL, LONGREAL, and BYTES.

EXTENSION 1517: Non-standard invocation of *identifier*.

This is caused by the Intel extension to standard Pascal that allows you to specify a second argument to REWRITE and RESET. Standard Pascal allows only one argument.

EXTENSION 1518: Non-standard indexed reference to component of a string constant.

Intel's Pascal-86 allows indexed references to components of a string constant. Standard Pascal does not support this feature.

EXTENSION 1520: Non-standard OTHERWISE clause in CASE statement.

Intel's Pascal-86 allows OTHERWISE clauses, which are not included in standard Pascal.

EXTENSION 1522: Character appears where string is expected.

Intel's Pascal-86 allows you to use a single-character constant wherever a string is permitted; i.e., a single-character constant can be interpreted as a PACKED ARRAY [1..1] OF CHAR. Standard Pascal does not allow a single-character constant where a string is permitted.

EXTENSION 1524: Factored procedural- or functional-parameter list.

In a parameter declaration, standard Pascal permits only one identifier to follow the PROCEDURE or FUNCTION word-symbol. Intel's Pascal-86 allows a list of identifiers.

EXTENSION 1525: Integer argument *n* of builtin *identifier* should be real.

Standard Pascal requires that the arguments to TRUNC and ROUND be of a real type.

EXTENSION 1534: Extended definition and compatability of strings.

Pascal-86 extends string type to include PACKED or UNPACKED arrays of type CHAR with a lower bound of 0 or 1.

EXTENSION 1535: Argument *n* of *identifier* is a member of a packed structure.

Pascal-86 will pass arguments that are components of a packed array of type CHAR.

WARNING 3102: Illegal character, "*char*" in control line.

The compiler found a character that is not permitted in an identifier while looking for a control name. The character is ignored, and compilation continues.

WARNING 3103: Premature end of control line.

The compiler found an end-of-line before finding a closing parenthesis of a control argument list. The control is not processed, and compilation continues.

WARNING 3104: Controls follow an INCLUDE control on the same line.

The INCLUDE control terminates the processing of the control line it is on. Any control that follows INCLUDE on the same control line is ignored, and compilation continues.

WARNING 3105: Control token, *control*, too long.

The characters scanned are discarded and scanning continues. WARNING 3201 will probably also occur.

WARNING 3106: Syntax error in *control* control; control will be ignored.

The *control* control is ignored, and compilation continues.

WARNING 3107: Syntax error at '<token>'; rest of control line has been ignored.

A syntax error was caused by the string of characters in < token >.

WARNING 3108: Constant *n* is greater than MAXLONGINT. MAXLONGINT is used.

The value specified falls outside the range of integer values. MAXLONGINT (+2147483647) is substituted, and compilation continues.

WARNING 3120: Domestic symbol *identifier* of subsystem *identifier* is declared public FOR module *identifier*, which is not a member of that subsystem.

The named symbol has not been exported from its subsystem, but its FOR clause includes a module that is not in this subsystem.

WARNING 3201: Unknown control, *control*.

The unknown control is ignored, and scanning continues.

WARNING 3202: Primary control, *control*, illegally specified outside invocation group.

You can specify primary controls only in the invocation line, or in the initial set of primary control lines (known as the "invocation group"). The control is ignored, and compilation continues.

WARNING 3206: Required parameter string missing from control *control*.

A TITLE or SUBTITLE control was specified without the title or subtitle. The control is ignored, and compilation continues.

WARNING 3208: Invalid SYMBOLSPACE parameter, default value used.

A valid parameter for the SYMBOLSPACE control ranges from 5 to 64.

WARNING 3209: Insufficient memory for SYMBOLSPACE allocation, maximum used.

The request for SYMBOLSPACE allocation is larger than the memory available; the maximum amount available is being used.

WARNING 3501: *identifier* does not match identifier on MODULE statement.

The name in the PROGRAM or PRIVATE statement does not match the name in the MODULE statement. The compiler assumes the PROGRAM or PRIVATE name to be the correct one, and compiles the module using that name.

WARNING 3505: Label *label* has been defined but not declared.

This warning occurs when a label that was not declared in a LABEL statement appears on a statement. The compiler assumes a declaration occurred and continues compiling, and the object code generated contains this repair, but the repair might not be what you intended for your program.

WARNING 3507: Label *label* has been declared but not defined.

This warning occurs when a label was declared in a LABEL statement but never used to label a statement in the block. The compiler ignores the label and continues compiling.

WARNING 3554: Multiple FORWARD declarations for *identifier*.

Multiple FORWARD declarations occurred for the same *identifier*. All FORWARD declarations after the first one are ignored, and compilation continues.

WARNING 3556: Definition and FORWARD declaration for *identifier* do not match.

This occurs if the parameter list or result value specified in a FORWARD definition does not agree with the corresponding parameter list or result type specification in the FORWARD declaration. The compiler chooses the specification in the FORWARD declaration, and continues compiling.

WARNING 3558: Multiple declarations for label *label*.

The compiler found more than one declaration for *label*. The compiler uses the first declaration, ignores the others, and continues compiling.

WARNING 3576: Program parameter *name* has not been declared a file.

Program parameters may be defined only as files within the program block. The explicit declaration stands, but the program parameter cannot be used as a file. Compilation continues with this repair, but the generated object code might not be what you intended for your program.

WARNING 3578: Argument *n* of *name* is a nested function or procedure.

Pascal-86 does not allow the argument to a BYTES parameter to be a nested procedure or function. If this procedure or function is invoked, all "up-level" references will be incorrect.

WARNING 3579: In argument *n* to FAR procedure or function *name*, attempt to pass a NEAR procedure as a BYTES argument.

WARNING 3580: Variant part of record is segmented.

It is unwise to declare part of a variant record to be greater than 64K. The data representation is not contiguous.

WARNING 3601: Insufficient memory to complete Cross-Reference.

WARNING 3902: Identifier, *identifier*, has been truncated to *identifier*.

Program and module identifiers may not exceed 31 characters. Public identifiers may not exceed 40 characters. (If the Debug control is active, local symbols are also truncated to 40 characters.)

WARNING 3906: Insufficient memory to generate type information for *identifier*.

A type record for the null type is generated for the specified identifier.

WARNING 3907: Overflow occurred when real constant
n was converted to binary. Infinity is used.

The value specified exceeds the largest representable real number. Infinity is substituted and compilation continues.

WARNING 3908: Underflow occurred when real constant
n was converted to binary. Zero is used.

The value specified is lower than the smallest representable real number. Zero is substituted and compilation continues.

WARNING 3909: Rounding occurred when real constant
n was converted to binary.

ERROR 5112: Multiple definitions for *identifier*
subsystem.

Each subsystem name must be unique. The segmentation control is ignored, and compilation continues.

ERROR 5114: Module *identifier* of *identifier* subsystem is also
claimed by *identifier* subsystem.

The named module appears in the HAS list for two or more segmentation controls. The module will remain in its originally specified subsystem.

ERROR 5115: Symbol *identifier* exported from *identifier*
subsystem is also claimed by *identifier* subsystem.

The named symbol appears in the EXPORTS list for two or more subsystem definitions. The symbol will remain in its originally specified subsystem.

ERROR· 5116: Symbol *identifier*, exported from identifier
subsystem, is actually declared in module *identifier*.

The named symbol appears in the EXPORTS list for the named subsystem, but is declared in a module belonging to another subsystem.

ERROR 5117: Predefined files, INPUT and OUTPUT,
exported from different subsystems.

The predefined files INPUT and OUTPUT can only be exported from the subsystem containing the main module of the program.

ERROR 5118: A HAS list is not permitted with an
open subsystem.

ERROR 5122: The open subsystem *identifier* has been
created; No more allowed.

The definition of the open subsystem must be the last segmentation control in the source program. The appearance of a segmentation control after a PUBLIC section not named in any HAS clause can cause this message.

ERROR 5123: The open subsystem *identifier* was created
for the module *identifier*; No more allowed.

ERROR 5203: Interrupt number, *n*, not in range
0..255. Using 0.

The interrupt number associated with an interrupt procedure in an INTERRUPT control is not in the proper range (0 to 255). The compiler uses a zero for the interrupt number, and compilation continues.

ERROR 5310: <ID>, a compiler-generated identifier, used to repair source.

Because < ID > was inserted by the error repair mechanism, an object module cannot be generated.

ERROR 5311: Nested IFs are not allowed.

ERROR 5312: Misplaced ELSEIF or ELSE control.

An ELSE or ELSEIF control must be specified in control line format—a dollar sign ($) in the left margin, followed by one or more controls, each separated by one or more blanks. Because ELSE and ELSEIF are generated controls, they may appear either in the command invocation line or on a control line located anywhere in the source code. Only one ELSE element may be included in an IF element.

ERROR 5313: Misplaced ENDIF control.

An ENDIF control must be specified in control line format—a dollar sign ($) in the left margin, followed by one or more controls, each separated by one or more blanks. Because ENDIF is a general control, it may appear either in the command invocation line or on a control line located anywhere in the source code.

ERROR 5314: Conditional control expression overflow.

ERROR 5315: Conditional control expression underflow.

ERROR 5316: Syntax error in identifier or number: *token*.

The expression following IF in an IF element must be one of these oeprators: OR, NOT, AND, <, ≤, =, >, ≥, and < >. The only operands allowed are switches and whole number constants from 0 to 255; otherwise, the incorrect identifier appears as *token*.

ERROR 5317: Number, *number*, must be in range 0 to 255.

ERROR 5500: Multiple declarations for *identifier*.

Multiple declarations were found for *identifier*. The compiler chooses the first declaration in the block. The extra declarations appear in the cross-reference listing showing no references to the object. The compiler continues compiling, but does not generate an object module.

ERROR 5502: Duplicate field name, *identifier*, in record definition.

The compiler adds the field to the record, but without a field name. The compiler continues compiling, but does not generate an object module.

ERROR 5504: Illegal circular definition of *identifier*.

A type or constant was defined using a bad recursive structure in which no pointers are used. For example, X=ARRAY[1..10] OF X, or Y=Z; Z=ARRAY[1..10] OF SET OF Y. The compiler supplies a neutral type and continues compiling, but does not generate an object module.

ERROR 5506: *identifier* has not been declared.

All identifiers must be declared. The compiler substitutes a neutral object for the illegal reference and continues compiling, but does not generate an object module.

ERROR 5510: *identifier* is not a constant as required.

An identifier on the right-hand side of a constant definition must denote a constant. The compiler supplies a neutral constant and continues compiling, but does not generate an object module.

ERROR 5511: *identifier* is not a numeric constant as required.

The identifier following a + or − sign in a constant definition must denote a numeric constant. The compiler substitutes a one for *identifier* and continues compiling, but does not generate an object module.

ERROR 5515: In the subrange specification *const1..const2*, the constants are not compatible.

The subrange bounds *const1* and *const2* are of different or incompatible types. The compiler nullifies the subrange specification and continues compiling, but it does not generate an object module.

ERROR 5517: The bounds of the subrange specification *const1*..*const2* are out of order.

The lower bound of the subrange is greater than the upper bound. The compiler nullifies the subrange specification and continues compiling, but it does not generate an object module.

ERROR 5519: The constant *identifier* is not legal in a subrange specification.

Pascal-86 requires that the endpoints of a subrange type be of an ordinal type. The compiler supplies a neutral type and continues compiling, but it does not generate an object module.

ERROR 5530: *identifier* is not a type as required.

The specified context requires a type identifier. The compiler supplies a neutral type and continues compiling, but it does not generate an object module.

ERROR 5534: Base type of set is not acceptable.

Pascal-86 requires that the base type of a set be of any ordinal type except WORD or LONGINT. The compiler supplies a neutral type and continues compiling, but it does not generate an object module.

ERROR 5535: Index type *identifier* is not acceptable.

Pascal-86 requires that the index type of an array specification be of an ordinal type. The compiler supplies a neutral type and continues compiling, but it does not generate an object module.

ERROR 5536: Tag type *identifier* is not acceptable.

Pascal-86 requires that the tag type of a variant record be of any ordinal type except LONGINT. The compiler supplies a neutral type and continues compiling, but it does not generate an object module.

ERROR 5537: *symbol* is a duplicate case constant in a variant record specification.

The case constant *symbol* is invalid, because it is a duplicate of another constant in the same record. The variant is incorporated into the record, but it is not associated with a case constant. The compiler continues compiling, but does not generate an object module.

ERROR 5538: Case constant *symbol* is incompatible with tag type.

The case constant *symbol* is invalid. The variant is incorporated into the record, but it is not associated with a case constant. Compilation continues, but an object module is not generated.

ERROR 5540: File type has a component that is or contains a file.

Pascal does not allow a component of a file to contain an imbedded file. The imbedded file is neutralized and compilation continues, but an object module is not generated.

ERROR 5542: The function result type *identifier* is not a scalar type.

ERROR 5544: The implicit variable associated with function *identifier* is never assigned a value in the body of the function.

ERROR 5547: Argument *n* of *identifier* is not addressable.

VAR arguments passed to SMALL procedures must be defined in SMALL subsystems, or must be obtained by dereferencing a SMALL pointer.

ERROR 5552: No definition for *identifier* appears after FORWARD declaration.

The *identifier* was declared as a FORWARD procedure, but a subsequent definition of *identifier* did not appear in the block, or the definition preceded the FORWARD declaration. The compiler continues compiling, but does not generate an object module.

ERROR 5560: Target of GOTO *label* is undefined.

The label referred to in the GOTO statement is not defined anywhere within an enclosing block. The compiler deletes the GOTO statement and continues compiling, but it does not generate an object module.

ERROR 5561: Target of GOTO *label* is inaccessible.

The label referred to in the GOTO statement is defined, but it is located on a statement at a nesting level that is deeper than the nesting of the GOTO, or otherwise incompatible with the nesting of the GOTO. The compiler deletes the GOTO statement and continues compiling, but it does not generate an object module.

ERROR 5562: Invalid call to function *identifier* in a procedure statement.

A function name was used by mistake in a procedure statement. The compiler ignores the procedure statement and continues compiling, but it does not generate an object module.

ERROR 5564: Variable expected in this context.

The left side of an assignment statement or an argument passed as a VAR parameter is not a variable, as required. The compiler ignores the statement, neutralizes its type, and continues compiling, but it does not generate an object module.

ERROR 5566: Right side of assignment statement is not compatible with left side.

The compiler ignores the assignment statement and continues compiling, but it does not generate an object module.

ERROR 5567: Illegal assignment to a variable that is or contains a file.

ERROR 5568: IF expression is not of type Boolean.

The expression following IF in an IF statement must be of type Boolean. The compiler ignores the expression and continues compiling, but it does not generate an object module.

ERROR 5570: CASE expression is not of an ordinal type.

The expression following CASE in a CASE statement must be of an ordinal type. The compiler ignores the expression, neutralizes the index type, and continues compiling, but it does not generate an object module.

ERROR 5574: Program parameter *name* has not been declared.

A program parameter was used as a file variable, but it was not declared. Program parameters other than INPUT and OUTPUT must be declared in the program block if they are to be used as file variables within the program. If the program tries another reference to it, another error will occur. Compilation continues, but no object code is generated.

ERROR 5576: Argument *n* of *identifier* is not memory resident.

The indicated argument cannot be passed as a VAR BYTES parameter. This error occurs when passing ordinal constants and string constants of length one.

ERROR 5577: WHILE expression is not of type Boolean.

The expression following WHILE in a WHILE statement must be of type Boolean. The compiler ignores the expression and continues compiling, but it does not generate an object module.

ERROR 5578: UNTIL expression is not of type Boolean.

The expression following UNTIL in an UNTIL statement must be of type Boolean. The compiler ignores the expression and continues compiling, but it does not generate an object module.

ERROR 5580: FOR loop index, *identifier*, is not a variable as required.

The index of a FOR loop must be a variable. The compiler ignores the reference, neutralizes its type, and continues compiling, but it does not generate an object module.

ERROR 5581: In argument *n* to procedure or function *name*, attempt to pass a procedure, variable, or memory-resident constant as a BYTES argument to a non-local procedure of a SMALL subsystem.

ERROR 5582: FOR loop index, *identifier*, is a global variable.

The index of a FOR loop must be defined in the immediately enclosing block. The compiler ignores the reference, neutralizes its type, and continues compiling, but it does not generate an object module.

ERROR 5584: The type of FOR loop index, *identifier*, is not acceptable.

Pascal-86 requires that a FOR-loop index be of an ordinal type. The compiler ignores the reference, neutralizes its type, and continues compiling, but does not generate an object module.

ERROR 5585: The initial-value expression of the FOR loop is incompatible with the type of the index variable.

The type of the index variable prevails, and the initial-value expression is ignored. The compiler continues compiling, but does not generate an object module.

ERROR 5586: The final-value expression of the FOR loop is incompatible with the type of the index variable.

The type of the index variable prevails, and the final-value expression is ignored. The compiler continues compiling, but it does not generate an object module.

ERROR 5587: Illegal reference to FOR loop index *identifier*.

This occurs when a FOR loop index appears on the left side of an assignment statement, as the variable argument in a function or procedure call, as the index of a nested FOR loop, or as the argument to READ or READLN. The compiler supplies a neutral object for *identifier* in this reference and continues compiling, but it does not generate an object module.

ERROR 5588: *identifier* is not a valid data reference in a WITH statement.

A variable used in a WITH statement must be a variable of a record type. The compiler ignores the reference and continues compiling, but it does not generate an object module.

ERROR 5592: The type of case constant, *symbol*, is not acceptable.

Pascal-86 requires that case constants be of any ordinal type except LONGINT. The case constant is ignored, but the statements of the case are analyzed as compilation continues. An object module is not generated.

ERROR 5594: *symbol* is a duplicate case constant.

Although the extra case constant is ignored, the CASE statements are analyzed and compilation continues. An object module is not generated.

ERROR 5595: Case constant, *identifier*, is incompatible with the CASE expression.

ERROR 5596: The variable *identifier...* is not a record as required by the WITH statement.

A variable used in a WITH statement must be a variable of a record type. The variable denotation beginning with *identifier...* does not denote such a variable. The reference is ignored and compilation continues, but an object module is not generated.

ERROR 6501: Compiler error in Semantic Analyzer (*n*, *m*).

Contact your supplier. (*n* and *m* are compiler debug information.)

ERROR 6502: Base type of set is not in the range of integers.

ERROR 6510: The type of the index expression does not match the index type of the variable *identifier...*.

The variable denotation beginning with *identifier...* denotes an array. The compiler ignores the index expression and selects an arbitrary component of the array. The compiler continues compiling, but it does not generate an object module.

ERROR 6512: The indexed variable *identifier...* is not an array.

The variable denotation beginning with *identifier...* is subscripted, but it does not denote an array. The compiler supplies a neutral object and continues compiling, but it does not generate an object module.

ERROR 6514: *identifier* is not a field of the designated record.

The compiler supplies a neutral object for *identifier* and .continues compiling, but it does not generate an object module.

ERROR 6516: The dereferenced variable *identifier...* is not a pointer.

The variable denotation beginning with *identifier...* contains the up-arrow (↑) operator, but it does not denote a pointer. The compiler supplies a neutral object and continues compiling, but it does not generate an object module.

ERROR 6518: The qualified variable *identifier...* is not a record.

The variable denoted by *identifier...* is qualified with a field designator, but the variable is not a record. The compiler supplies a neutral object and continues compiling, but it does not generate an object module.

ERROR 6520: *identifier* cannot be referenced in an expression.

The *identifier* is not a constant, variable, or function, and therefore it cannot be referred to in an expression. The compiler supplies a neutral object and continues compiling, but it does not generate an object module.

ERROR 6533: Function reference *identifier* references the return value.

ERROR 6540: For the operator *symbol*, type of operand is incompatible with operator.

A wrong type of operand was used. The compiler supplies a neutral object for the operator and continues compiling, but it does not generate an object module.

ERROR 6541: The operands of operator *symbol* are incompatible with each other.

The compiler supplies a neutral result for the operation and continues compiling, but it does not generate an object module.

ERROR 6542: Illegal call to procedure *identifier* in expression.

A procedure name was used by mistake in an expression. The compiler supplies a neutral object for the call and continues compiling, but it does not generate an object module.

ERROR 6544: Argument list in call to *identifier* is too short.

The compiler supplies a neutral object for the call and continues compiling, but it does not generate an object module.

ERROR 6545: Near symbol *symbol*, a set expression element is not an ordinal value.

The compiler deletes the element and continues compiling, but it does not generate an object module.

ERROR 6546: Near symbol *symbol*, type of set expression element is incompatible with expressions that follow it.

The compiler deletes the incompatible element, and continues compiling, but it does not generate an object module.

ERROR 6548: Argument list in call to *identifier* is too long.

The compiler supplies a neutral object for the call and continues compiling, but it does not generate an object module.

ERROR 6550: Argument *n* of *identifier* is not the same type as the corresponding VAR parameter.

The *n*th parameter of *identifier* is a VAR parameter. The compiler supplies a neutral object for the *n*th argument in the call to *identifier* and continues compiling, but it does not generate an object module.

ERROR 6552: Argument *n* of *identifier* is not assignment-compatible with the corresponding value parameter.

The *n*th parameter of *identifier* is a value parameter. The compiler supplies a neutral òbject for the *n*th argument in the call to *identifier* and continues compiling, but it does not generate an object module.

ERROR 6555: Argument *n* of *identifier* is not a function or procedure.

The *n*th parameter of *identifier* is a functional or procedural parameter. The compiler supplies a neutral object for the *n*th argument in the call to *identifier* and continues compiling, but it does not generate an object module.

ERROR 6556: The parameter list of argument *n* of *identifier* does not match the parameter list of the corresponding procedural parameter.

The *n*th parameter of *identifier* is a procedural or functional parameter. The compiler supplies a neutral object for the *n*th argument in the call to *identifier*.and continues compiling, but it does not generate an object module.

ERROR 6557: *identifier* is not eligible as an interrupt procedure.

To be allowed as an interrupt procedure, *identifier* cannot have any parameters, and it must be defined at level one. The compiler ignores the definition and continues compiling, but it does not generate an object module.

ERROR 6558: Second parameter of SETINTERRUPT is not an interrupt procedure.

The second parameter for the SETINTERRUPT procedure must be an interrupt procedure. The compiler ignores the statement and continues compiling, but it does not generate an object module.

ERROR 6559: Argument *n* of *identifier* is an interrupt procedure, which cannot be a procedural argument.

ERROR 6560: *identifier* is not a procedure as required.

The context requires that *identifier* be a procedure. The compiler deletes the statement and continues compiling, but it does not generate an object module.

ERROR 6561: Attempt to reassign interrupt number *n* to interrupt procedure *identifier*.

Pascal does not allow the same interrupt number to be assigned to more than one interrupt procedure.

ERROR 6562: Argument *identifier* is a predefined routine; it cannot be a procedural argument.

Pascal-86 does not allow you to pass predefined routines as arguments to user-defined routines. The compiler supplies a neutral object for *identifier* and continues compiling, but it does not generate an object module.

ERROR 6564: Argument *n* of *identifier* must be a variable.

An expression was used as an argument for a VAR parameter where a variable must be used. The compiler supplies a neutral object for the argument and continues compiling, but it does not generate an object module.

ERROR 6565: Argument *n* of *identifier* cannot be a member of a packed structure.

Components of packed structures cannot be passed as VAR arguments. The compiler supplies a neutral object for the argument and continues compiling, but it does not generate an object module.

ERROR 6566: Argument *n* of *identifier* is an invalid argument specification.

The argument notation e:e or e:e:e can be used only in the predefined procedures WRITE and WRITELN. The compiler ignores the extra notation and continues compiling, but it does not generate an object module.

ERROR 6567: Format of argument *n* of *identifier* is invalid for non-real argument.

The argument notation e:e:e can be used only for REAL type arguments to the predefined procedures WRITE or WRITELN. The compiler ignores the notation and continues compiling, but it does not generate an object module.

ERROR 6568: The field width specifier in argument *n* of *identifier* must be an integer value.

The *identifier* is either WRITE or WRITELN.

ERROR 6570: Standard file *filename*, implied by call to predefined I/O routine, has not been defined.

In a call to EOF, EOLN, READ, WRITE, READLN, WRITELN, or PAGE, a file parameter was omitted, implying one of the standard files INPUT or OUTPUT; however, neither standard file appeared as a program parameter. This error can occur only in a main module. The compiler assumes that INPUT and OUTPUT are program parameters if they are referenced in a non-main module. The compiler assumes that the call implies one of the standard files and continues compiling, but it does not generate an object module.

ERROR 6572: Argument *n* of *identifier* is not a Text file as required.

The file argument to READ, WRITE, READLN, WRITELN, EOLN, and PAGE must be a text file. The compiler continues compiling, but it does not generate an object module.

ERROR 6580: Argument *n* of *identifier* is not a constant.

The forms of NEW and DISPOSE for variant records must have constant values for variant selectors. The compiler deletes the statement and continues compiling, but it does not generate an object module.

ERROR 6582: Argument *n* of *identifier* is incompatible with the tag type of the variant record to which it corresponds.

The *n*th argument of either NEW or DISPOSE is incompatible. The compiler deletes the statement and continues compiling, but it does not generate an object module.

ERROR 6583: Argument *n* of *identifier* is not an array.

The *n*th argument in a call to PACK or UNPACK is not an array as required. The compiler deletes the statement and continues compiling, but it does not generate an object module.

ERROR 6584: The array arguments of *identifier* are incompatible.

The array arguments for the PACK and UNPACK procedures are described in section 8.6. The compiler deletes the statement and continues compiling, but it does not generate an object module.

ERROR 6586: The index argument of *identifier* is incompatible with the index type of the arrays.

The *identifier* is either PACK or UNPACK.

ERROR 6588: Argument *n* of *identifier* is incompatible with the component type of the file.

In a call to READ, WRITE, READLN, or WRITELN, an argument is not compatible with the component type of the file. For example, in "READ(f,c)", this error would occur if "c" is incompatible with "f↑". The compiler deletes the statement and continues compiling, but it does not generate an object module.

ERROR 6589: Argument *n* of PACK or UNPACK is not an array.

ERROR 6801: Integer overflow exception detected at compile-time.

ERROR 6802: Range check exception detected at compile-time.

LIMIT EXCEEDED 7301: At most 10 switches allowed for conditional compilation.

LIMIT EXCEEDED 7501: The definition of *identifier* requests more than 65535 bytes.

A data structure cannot have more than 65535 bytes. Offsets generated for the data structure are invalid. The compiler continues compiling, but it does not generate an object module.

LIMIT EXCEEDED 7502: The definition of *symbol* requests more than 65535 bits.

A packed data structure cannot have more than 65535 bits (8192 bytes). Offsets generated for the data structure are invalid. The compiler continues compiling, but it does not generate an object module.

LIMIT EXCEEDED 7503: The definition of the set requests more than 32767 elements.

Sets cannot have more than 32767 elements. Offsets generated for the data structure are invalid. The compiler continues compiling, but it does not generate an object module.

LIMIT EXCEEDED 7504: The range of values spanned by the constants of a CASE statement has more than 1009 values.

The CASE statement jump table cannot have more than 1009 entries. The compiler ignores the extra entries and continues compiling, but it does not generate an object module.

LIMIT EXCEEDED 7505: The data segment of *identifier* exceeds 65535 bytes.

LIMIT EXCEEDED 7507: The size of the dynamic area requested exceeds 65535 bytes.

LIMIT EXCEEDED 7508: Local variables cannot be segmented.

LIMIT EXCEEDED 7509: Segmented data is only allowed in the LARGE model.

LIMIT EXCEEDED 7510: Segmented data cannot be passed as value parameters.

LIMIT EXCEEDED 7511: File variables cannot be segmented.

LIMIT EXCEEDED 7512: Too many extra segments required.

LIMIT EXCEEDED 7513: Dynamic variables may not contain nested segmented data.

LIMIT EXCEEDED 7514: Allocation of segmented arrays is only allowed in the LARGE model.

LIMIT EXCEEDED: 7901: Too much debug information.

FATAL ERROR 8201: Maximum INCLUDE nesting level (5) exceeded.

When including a file that also includes a file, etc., you can include only up to a total of five files. Compilation is aborted.

FATAL ERROR 8301 to 8303: Parse stack/buffer limit exceeded.

You may be able to avoid a stack overflow by reducing the nesting of procedures and shortening the length of statement lists. Compilation is aborted.

LIMIT EXCEEDED 8501: Semantic analyzer stack/buffer overflow.

You may be able to avoid a stack overflow by reducing the nesting of procedures. Compilation is aborted.

LIMIT EXCEEDED 8503: Memory exhausted after *n* bytes.

The storage required during semantic analysis is exhausted, and compilation is aborted. Use the Symbol Space control to increase allocated memory. You may be able to avoid this overflow by shortening the length of statement lists.

LIMIT EXCEEDED 8801: Expression too complex.

Expressions cannot have more than 50 operands, and expression nesting is also limited. Rewrite the expression to reduce its complexity. Compilation is aborted. Adding more memory or increasing the Symbol Space parameter may help.

LIMIT EXCEEDED 8802: *type* segment of module exceeds 65535 bytes.

The *type* of the segment is either CODE or STACK. Compilation is aborted.

LIMIT EXCEEDED 8803: Compiler generated label table overflow.

There are too many flow control statements (IF-THEN, WHILE, CASE, etc.) in the current procedure. Break the procedure into smaller procedures.

LIMIT EXCEEDED 8901: Object module too complex.

LIMIT EXCEEDED 8902: *identifier* segment of module exceeds 65535 bytes.

FATAL ERROR 9000: I/O error on *filename*: *system information*

The compiler cannot find or open *filename*. Compilation is aborted and control is returned to the operating system. Check to see if the file exists.

FATAL ERROR 9001: Unable to open INCLUDE file
filename: *system information*

The INCLUDE file is not present, or is already open. Compilation is aborted, and control is returned to the operating system.

FATAL ERROR 9002: I/O error on *filename*: *system information*

The compiler cannot find or open *filename*. Compilation is aborted and control is returned to the operating system. Check to see if the file exists.

FATAL ERROR 9005: Input file missing or syntax
error in invocation line.

FATAL ERRORS 9006 to 9017: I/O error on compiler
work file: *system information*

The compiler cannot open one or more of the eight workfiles for some reason. Check to see if the files are assigned to another disk drive besides the designated workfile drive, or if the proper disk is not in the designated workfile drive. Use the WORK command to change the designated workfile drive. If the error persists, contact your supplier. Compilation is aborted, and control is returned to the operating system.

FATAL ERROR 90*xx*: Compiler error in Root.

Unexpected condition; contact your supplier. Compilation is aborted. (*xx* represents a particular number useful only to your supplier.)

FATAL ERROR 9020: Compiler error in U.D.S.M.: Not
enough memory.

FATAL ERROR 9021: Compiler error in U.D.S.M.: Not
enough disk space.

FATAL ERROR 9022: Compiler error in U.D.S.M.: Bad
object.

FATAL ERROR 9023: U.D.S.M. System error: *system
information*

FATAL ERROR 9201: Unknown control, *control*, in
invocation group.

The "invocation group" includes the invocation line and the initial set of primary control lines in the source file. Compilation is aborted.

FATAL ERROR 9202: Missing ENDIF control.

FATAL ERROR 92*xx*: Compiler error in scanner.

Compilation is aborted. Contact your supplier.

FATAL ERROR 93*xx*: Compiler error in parser.

Compilation is aborted. Contact your supplier.

FATAL ERROR 9501: Compiler error in Semantic
Analyzer (*n*, *m*).

Compilation is aborted. Contact your supplier. (*n* and *m* are compiler debug information.)

```
FATAL ERROR 9599: Compiler error in Semantic
Analyzer (n).
```

Compilation is aborted. Contact your supplier. (*n* is compiler debug information.)

```
FATAL ERROR 95xx: Compiler error in Semantic
Analyzer.
```

Compilation is aborted. Contact your supplier.

```
FATAL ERROR 96xx: Compiler error in Cross-Reference
Generator.
```

Compilation is aborted. Contact your supplier.

```
FATAL ERROR 98xx: Compiler error in Code Generator.
```

Compilation is aborted. Contact your supplier.

```
FATAL ERROR 99xx: Compiler error in Object Module
Generator.
```

Compilation is aborted. Contact your supplier.

```
FATAL ERROR 9902: Compiler error in Object Module
Generator (n).
```

```
FATAL ERROR 9904: Compiler-generated label address
error.
```

CHAPTER 14
RUN-TIME EXCEPTIONS

This chapter describes all of the run-time exceptions (errors) that are handled by the run-time software unique to Pascal-86.

Run-time messages are coded by number and are listed in this chapter in numeric order by their code numbers so that you can easily look up any message you received.

A masked floating-point run-time error can occur without stopping the program. When a run-time error other than a masked floating-point error occurs, the default Pascal-86 run-time system stops running the program, prints a run-time exception message, and returns control to the operating system.

There are three types of run-time exceptions: non-floating-point run-time exceptions (14.1), floating-point function exceptions (14.2), and floating-point 8087 exceptions (14.2).

14.1 Run-Time System Exceptions

There are several types of non-floating-point run-time exceptions: I/O, operating environment, and range exceptions, to name a few.

Run-time system exception messages take the following form:

```
*** RUN-TIME type EXCEPTION: code
*** NEAR LOCATION hhhhH:hhhhH
*** JOB ABORTED.
```

The *type* of the run-time exception can be one of the following types:

```
PASCAL I/O
I/O
OPERATING ENVIRONMENT
INTEGER ZERO DIVIDE
INTEGER OVERFLOW
RANGE
CHECK
PASCAL SET
```

For each type, the *code* is the hexadecimal exception code number for each message. (If no type is given, refer to the exception conditions for your specific operating system.) The hexadecimal locations *hhhh*H:*hhhh*H are the values in CS:IP after control returns from the run-time system to the program. Each message is described in the subsequent sections by type and by code number.

14.1.1 Input/Output Exceptions

```
RUN-TIME PASCAL I/O EXCEPTION: 1101H
```

An attempt to open a file was not successful. The file may not exist.

```
RUN-TIME PASCAL I/O EXCEPTION: 1102H
```

The first format specifier was negative or zero, but not positive as required.

```
PASCAL I/O EXCEPTION: 1103H
```

The second format specifier was negative or zero, but not positive as required.

```
RUN-TIME PASCAL I/O EXCEPTION: 1104H
```

An input operation was attempted on a file opened for output.

```
RUN-TIME PASCAL I/O EXCEPTION: 1105H
```

An output operation was attempted on a file opened for input.

```
RUN-TIME PASCAL I/O EXCEPTION: 1106H
```

The record number on a SEEKREAD or SEEKWRITE is negative.

```
RUN-TIME PASCAL I/O EXCEPTION: 1108H
```

Attempt to open a text file for random I/O.

```
RUN-TIME PASCAL I/O EXCEPTION: 1109H
```

A Random I/O operation was attempted on a non-random file.

```
RUN-TIME PASCAL I/O EXCEPTION: 110AH
```

The ENDPOSITION function was called with an empty file.

```
RUN-TIME PASCAL I/O EXCEPTION: 110BH
```

A SEEKWRITE was attempted beyond the end of the file.

```
RUN-TIME PASCAL I/O EXCEPTION: 1110H
```

SETRANDOM was called with a file that already was opened.

```
RUN-TIME I/O EXCEPTION: 9102H
```

The end of a file was encountered when illegal.

```
RUN-TIME I/O EXCEPTION: 9103H
```

The integer field on input does not conform to the Pascal signed decimal integer syntax.

```
RUN-TIME I/O EXCEPTION: 9104H
```

The floating-point field on input does not conform to the Pascal run-time signed number syntax.

```
RUN-TIME I/O EXCEPTION: 9105H
```

The integer field on text file input defined a signed integer which could not fit into the INTEGER range (−32767 to +32767).

```
RUN-TIME I/O EXCEPTION: 9106H
```

The integer field on text file input defined a signed integer that could not fit into the LONGINT range (−2,147,483,647 to +2,147,483,647).

```
RUN-TIME I/O EXCEPTION: 9107H
```

The floating-point field on text file input defined a signed number that was too large (overflow) to fit into the TEMPREAL range (-2^{16384} to 2^{16384}).

```
RUN-TIME I/O EXCEPTION: 9108H
```

The floating point field on text file input defined a signed number that was too small (underflow) to fit into the TEMPREAL range (-2^{-16383} to 2^{-16383}).

```
RUN-TIME I/O EXCEPTION: 9109H
```

The integer field or text file output defined a signed integer that could not fit into the BYTE range (−127 to +127).

14.1.2 Operating Environment and Heap Exceptions

```
RUN-TIME OPERATING ENVIRONMENT EXCEPTION: 1300H
```

Programs that use floating-point functions must be linked to the run-time library CEL87.LIB. If your program is linked to EH87.LIB but not to CEL87.LIB, this exception will be reported when the first floating-point function is accessed.

```
RUN-TIME OPERATING ENVIRONMENT EXCEPTION: 1501H
```

Invalid file preconnection syntax on the program's command line.

```
RUN-TIME HEAP EXCEPTION: 1151H
```

The pointer passed to the DISPOSE function had an illegal value. An illegal value occurs if the pointer was not initialized, was assigned a NIL value, was already DISPOSEd, or was not returned by NEW.

```
RUN-TIME HEAP EXCEPTION: 1152H
```

There is insufficient free memory available to fill the request.

```
RUN-TIME HEAP EXCEPTION: 1153H
```

The SMALL heap size returned by TQGETSMALLHEAP is less than 16 bytes.

14.1.3 Integer Exceptions

```
RUN-TIME INTEGER ZERO DIVIDE EXCEPTION: 8000H
```

There was an attempt to divide by zero.

```
RUN-TIME INTEGER OVERFLOW EXCEPTION: 8001H
```

A signed integer overflow occurred.

14.1.4 Set Exceptions

```
RUN-TIME PASCAL SET EXCEPTION: 1131H
```

A set on the stack could not be represented in memory. Sets cannot have more than 32767 elements.

```
RUN-TIME PASCAL SET EXCEPTION: 1132H
```

The stack overflowed on a set operation. Either the set is too big or the set operation is too complex.

```
RUN-TIME PASCAL SET EXCEPTION: 1133H
```

A set with a size of zero was used in a set function.

```
RUN-TIME PASCAL SET EXCEPTION: 1134H
```

An attempt was made to add an invalid member to a set. All set elements must be compatible with the base type of the set.

```
RUN-TIME PASCAL SET EXCEPTION: 1135H
```

The result of a union operation was too large.

14.1.5 Compiler Range and Check Errors

```
RUN-TIME RANGE EXCEPTION: 8006H
```

```
RUN-TIME CHECK EXCEPTION: 8017H
```

The program was compiled with the CHECK control. At run time, the compiled code checks for out-of-range assignments, out-of-range array subscripts, stack and integer overflow, and invalid pointer references. These conditions acknowledge extra checks the compiler generated in-line. The run-time check exception may also occur when the appropriate 8087 interface library was omitted from the program link list.

14.2 Floating-Point Function Exceptions and 8087 Exceptions

Two kinds of real (floating-point) exceptions may occur: those resulting from the execution of predefined floating-point functions, and those resulting directly from floating-point arithmetic operations performed by the 8087 Numeric Data Processor or its emulator.

Floating-point function exception messages take the following form:

```
*** RUN-TIME FLOATING-POINT function EXCEPTION status
*** NEAR LOCATION hhhhhH
*** JOB ABORTED.
```

The *function* can be one of the following:

```
EXP
LN
SIN
COS
TAN
ARCSIN
ARCCOS
ARCTAN
TRUNC
ROUND
INT
NINT
```

The *status* is the hexadecimal value of the 8087 STATUS register, and the location *hhhhh* is the 20-bit physical address of the location of the exception. The 8087 STATUS values are described in the *iAPX 86,88 User's Manual.* General floating-point exceptions are discussed in the next section.

Floating-point 8087 exception messages take the following form:

```
*** RUN-TIME 8087 EXCEPTION status
*** INSTR OPCODE op
*** MEMOP ADDRESS hhhhhH
*** NEAR LOCATION hhhhhH
*** JOB ABORTED.
```

The *status* is the hexadecimal value in the 8087 STATUS register. The *op* is the hexadecimal value of the 8087 instruction opcode register. The *hhhhh*H is a hexadecimal 20-bit physical address. The 8087 registers are described in the *iAPX 86,88 User's Manual.*

As discussed in 7.1.8, there are six possible real arithmetic error, or exception, conditions: invalid operation, denormalized operand, zero divide, overflow, underflow, and precision.

This section first discusses the meaning of the six types of exceptions, what conditions may cause them, and the actions performed by the chip or emulator when the exceptions occur with the corresponding exception mask bits set; i.e., with the exceptions masked (The *iAPX 86,88 User's Manual* discusses the unmasked case.)

Following this are explanations of rounding, denormalized and unnormalized numbers, unnormalized arithmetic, and infinity arithmetic. These discussions should suffice for Pascal-86 users. However, if you are also writing modules in other languages to interface with the 8087 chip or emulator, you may wish to see the *iAPX 86,88 User's Manual* for a fuller explanation of some topics.

NOTE

Pascal-86 presets certain 8087 modes (explained in the *iAPX 86,88 User's Manual*) to the following recommended settings:

- The infinity arithmetic mode is Projective.
- The rounding mode is Round-to-Nearest.
- The precision mode for intermediate results is 64 bits.
- All 8087 exception conditions are masked except Invalid Operation, which is unmasked.
- The 8087 interrupt enable mask bit is zero (interrupt enabled).

You cannot change the infinity arithmetic, rounding, or precision mode settings from a Pascal-86 program. You may, however, change the exception mask bits or the interrupt enable mask bit using the MASK8087ERRORS procedure (8.10.2). The following discussions assume that you have not changed any of these settings. If you use any of the functions, SIN, COS, TAN, ARCSIN, ARCTAN, EXP, LN, ROUND, or LROUND, you cannot unmask the precision error. The precision exception bit in the 8087 STATUS word becomes undefined upon return from these functions.

The exception descriptions here refer to some topics not yet discussed, such as denormalized numbers, unnormalized numbers, rounding, and infinity arithmetic. For explanations of these items, see 14.2.1.

Invalid Operation

An *invalid operation* exception occurs when either an operand is invalid for the specified operation, or the operation itself is invalid. This exception generally indicates a program error; so even if you mask all other exceptions, it is recommended that you leave Invalid Operation unmasked. An Invalid Operation exception is signalled when any one of the following conditions occurs:

- One or more of the operands is a trapping NaN.
- One or more of the operands in the computation sequence was unnormalized or denormalized, and the result cannot be guaranteed, because significant information was lost. (Not all operations on unnormalized or denormalized numbers result in loss of significant information; those that do not will not signal Invalid Operation.)
- Any of the following operations is attempted: infinity+infinity, infinity−infinity, 0.0*infinity, infinity*0.0, infinity/infinity, 0.0/0.0, valid number/unnormalized number, valid number/denormalized number, or 0.0/pseudo-zero (a special representation, described in the *iAPX 86,88 User's Manual*).
- In TRUNC or ROUND, the operand is too large to fit into the INTEGER format.
- In LTRUNC or LROUND, the operand is too large to fit into the LONGINT format.
- In comparisons via any of the relational operators $<$, $\leq$, $>$, or $\geq$, infinity is compared to some value other than infinity.

The following are specific cases that cause invalid operation exceptions:

- SQRT (x) where x is a negative number, denormal number, unnormal number, or ± infinity (in projective mode).
- SIN (x), COS (x), TAN (x) where x is ± infinity, or $|x| \geq 2^{-63}$ and unnormal
- ARCSIN (x), ARCCOS (x) where x is ± infinity, $|x| \geq 2^{-63}$ and unnormal, or $|x| > 1$
- ARCTAN (x) where $|x| \geq 2^{-63}$ and unnormal.
- EXP(x) where x is ± infinity (in projective mode) or $|x| \geq 2^{-63}$ and unnormal.
- LN(x) where x is a negative number, a denormal number (and the Denormalized Exception mask, AT87DENR, is set), ± infinity (in projective mode), or $|x| \geq 2^{-63}$ and unnormal.

If the Invalid Operation exception is masked, the result (if a real result is expected) is a NaN. For a comparison, the result is unordered.

Denormalized Operand

This exception arises when one or more of the operands is a denormalized number. This could occur if you used uninitialized data or if a masked underflow exception occurred in a previous operation. If this exception is masked, correct unnormalized arithmetic is performed, as described in 14.2.1.

Zero Divide

In a division operation, if the divisor is a normal zero and the dividend is a finite nonzero number, then the Zero Divide exception occurs. If this exception is masked, the result is infinity. LN(0) and TAN of PI/2 also cause zero divide exceptions.

Overflow

If a rounded result is finite but its exponent is too large to represent in the REAL format, the Overflow exception occurs. If this exception is masked, an overflow yields infinity, and the Precision exception also occurs.

Underflow

The Underflow exception occurs when either of the following conditions arises:

- A rounded result has too small an exponent to be represented in the REAL format without normalizing.
- An intermediate product or quotient, where neither operand is a normal zero, is indistinguishable from a normal zero. (This cannot occur with normalized operands.)

If the Underflow exception is masked, the result is a correctly rounded denormalized number or zero.

Precision

If the correctly rounded result of an operation is not the same as the unrounded value, the Precision exception occurs. If this exception is masked, no special action is performed; the correctly rounded result is delivered.

14.2.1 Floating-Point Topics

The following section explains topics mentioned in the exception discussions just given. If you are using Pascal-86 only, this section should provide the background you need. If you are changing the 8087 modes preset for Pascal-86, see the *iAPX 86,88 User's Manual.*

Rounding

In Pascal-86, all implicit rounding is done in Round-to-Nearest mode. In this mode, the operand is rounded to the nearest representable value, or to the nearest even number in case of a tie. The rounding mode determines the sign assigned to zero: if x is finite, then x-x=x+(−x)=+0 in Round-to-Nearest mode. However, x+x=x−(−x) always has the same sign as x even when x is zero.

Normalized, Denormalized, and Unnormalized Numbers

In a *normal zero*, the exponent is zero and all significant bits are zero. A value is *normalized* if it is a normal zero, or if the leading bit of the significand is one and the exponent is greater than zero. A *denormalized* number is one that has a zero exponent and a zero explicit or implicit leading bit, but is not a normal zero. An *unnormalized* number is one that has an exponent greater than zero and a zero explicit leading bit (the term *unnormalized* applies only to numbers in the TEMPREAL format).

Note that the 8087 does not perform normalized arithmetic; to implement the IEEE Standard normalized mode of arithmetic, you must link in the EH87.LIB support library (see 12.2.2) and unmask the D exception. If the library is linked in but the D exception stays masked, normalized arithmetic will not be implemented.

When all operands have been normalized, the operations are performed as if the precision were infinite, before rounding occurs as described in the previous section. The following section describes the result if one or more of the operands are not normalized. If one operand is a NaN, the result, if any, is that NaN; and if more than one operand is a NaN, the result, if any, is the NaN that is largest in magnitude.

Unnormalized Arithmetic

Due to the nature of computer arithmetic on very large or very small numbers, there are some cases in which information will be lost. The 8087 processor and its emulator are designed to preserve as much information as possible from a computation, even when all of the information cannot be saved. Unnormalized arithmetic performed by Pascal-86 programs serves this purpose. When unnormalized numbers appear during a computation sequence *and* generate more unnormalized numbers rather than disappearing immediately, their presence indicates that some information has been lost (that is, greater precision could not be guaranteed), but their values still give some information about the computational results.

The following rules apply when at least one operand is not normalized, provided the Invalid Operation exception does not occur. They specify when normalization is to occur and the resulting exponent value if normalization does not occur. Rounding and the handling of overflow and underflow are performed after the assignments shown below. Such rounding and overflow/underflow handling may modify the results. In the following, *x* and *y* are real expressions, and *expon(x)* refers to the exponent of *x*.

Assignment (z:=x): expon(z)=expon(x).

Add/subtract (z:=x±y): Let m=max(expon(x),expon(y)). If at least one of the operands having exponent m is normalized, then z is normalized before rounding. Otherwise expon(z)=m.

Multiply (z:=x*y): expon(z)=expon(x)+expon(y).

Divide (z:=x/y): expon(z)=expon(x)−expon(y)−1 when y is normalized and nonzero.

Integer part (z:=the temporary real value obtained by applying the Round-to-Nearest rule to x): if expon(x) is so large that x must already be an integer regardless of its significand bits, then z is identically x. Otherwise, z is normalized.

Comparison Comparisons are made as if both operands had been normalized first.

Infinity Arithmetic

The representation of infinity is a temporary real value that behaves like an infinite value in real computations. In the infinity arithmetic mode used in Pascal-86 (Projective mode), +infinity=−infinity.

The sign bit of the product or quotient of two real numbers is the exclusive OR of the operands' sign bits, even when the operand is zero or infinity.

APPENDIX A
DIFFERENCES BETWEEN PASCAL-86 AND OTHER VERSIONS OF PASCAL

Pascal-86 conforms to the ANSI/IEEE770X3.97–1983. However, it may differ slightly from other Pascal dialects you have been using. The various Pascal implementations in use differ primarily where weaknesses in Pascal have been discovered; the Intel extensions that are part of Pascal-86 are examples of attempts to make up for these weaknesses. Implementations also differ where there is ambiguity in Jensen and Wirth's *Pascal User Manual and Report.*

This appendix provides information about such differences in order to help you use existing Pascal programs, originally written for other systems, in your iAPX 86 and iAPX 88 microcomputer applications. This information indicates what parts of your programs need to be modified before you compile, link, locate, and run them as Pascal-86 programs. The first part of this appendix also lists the extra Pascal-86 features (extensions) you may use to improve your programs.

NOTE

The *Pascal-80 User's Guide* and some other publications refer to the Pascal of Jensen and Wirth's *Pascal User Manual and Report* as "standard Pascal." This is true because until the recent standardization effort, Jensen and Wirth Pascal was the de facto standard. In the manual you are reading, "standard Pascal" refers to the ISO *Draft Proposal.*

A.1 Intel Extensions to Standard Pascal

This section lists features in Pascal-86 that are not part of standard Pascal as specified in the ANSI/IEEE770X3.97–1983. Following each listed feature is a reference to the chapter or section of this manual where the feature is described.

Most of these extensions are flagged by the compiler when the NOEXTENSIONS control (10.3.7) is active. However, those marked by an asterisk * are not flagged under NOEXTENSIONS.

A.1.1 Major Extensions

- Separate compilation facilities, including module headings, interface specifications, and headings (4.2.2, 4.2.3, 4.2.4)
- The predefined types WORD, LONGINT, LONGREAL, and TEMPREAL (5.3.1) and the built-in functions LORD, WRD, LTRUNC, and LROUND (8.1, 8.4)
- The port input and output procedures INBYT, INWRD, OUTBYT, and OUTWRD (8.8)
- The interrupt control procedures SETINTERRUPT, ENABLEINTERRUPTS, DISABLEINTERRUPTS, and CAUSEINTERRUPT (8.9)

A.1.2 Minor Extensions

- Interpretation of the tab character as a logical blank (3.2)*
- Octal-, binary-, and hexadecimal-based integer constants (3.3.2)

- Real constants outside the range of REAL values (3.3.3)*
- Labels larger than 9999 (3.3.4)
- String constants continued across input record (input line) boundaries (3.3.5)
- Single-character constants used as type PACKED ARRAY [1..1] OF CHAR (3.3.5, 5.3.2)
- Indexed references to components of a string constant (5.2)
- Use of an identifier, including a procedure or function identifier, before it is defined (5.3, 6.5)*
- The predefined CHAR constants CR (carriage return) and LF (line feed) (5.3.1, 5.3.2)
- In a variant record, case constants that do not map onto the range of the tag type (5.3.2)
- Factored procedural and functional parameter lists; that is, more than one identifier following the PROCEDURE or FUNCTION keyword in the parameter list syntax (6.4.1)
- Variable parameters of the predefined type BYTES (6.4.4)
- Specification of a parameter list (and result type, for a function) in the heading of a procedure or function previously declared to be FORWARD (6.5)
- Ordinal Type Transfer Functions (with extended types) (Chapter 7 and 8).
- Support of the proposed IEEE standard for floating-point arithmetic (7.1.8, 14.6, 14.7)*
- OTHERWISE clause in CASE statement (7.2.5)
- The arithmetic functions TAN, ARCSIN, and ARCCOS (8.3)
- Second argument to RESET and REWRITE, to designate a physical file (8.7.1, 8.7.2)
- File preconnection facility (8.7.1, 8.7.2, 12.4.1)*
- More lenient format for real number input to READ and READLN (8.7.6)*
- READ and READLN input of plus infinity, minus infinity, and the indefinite NaN (8.7.6)*
- WRITE and WRITELN output of plus infinity, minus infinity, and NaN's (8.7.6)*
- The 8087 procedures GET8087ERRORS and MASK8087ERRORS (8.10)
- The predefined types AT87ERRORS and AT87EXCEPTIONS and the predefined constants AT87NVLD, AT87DENR, AT87ZDIV, AT87OVER, AT87UNDR, AT87PRCN, AT87RSVD, and AT87MASK (8.10)
- All compiler controls in the source text (Chapter 10) (Only general controls will be flagged under NOEXTENSIONS; the initial set of control lines may be considered the logical extension of the compiler invocation line, and therefore not part of the Pascal program per se.)

A.2 Differences Between UCSD Pascal (Pascal-80) and Standard Pascal

UCSD Pascal is a commonly used Pascal dialect; Intel's interpreted Pascal-80 language is an implementation of UCSD Pascal. UCSD Pascal differs from Pascal-86 in a number of ways, most of them are differences between UCSD Pascal and standard Pascal. The following list explains the features that differ. The section number reference following each description of a Pascal-86 feature denotes the section of this manual which gives further information.

- In UCSD Pascal, the comment bracket symbol { must match the bracket symbol } , and (* must match *). In Pascal-86, the matching of comment brackets follows standard Pascal: either left bracket symbol matches either right bracket symbol. (3.2.1)
- In UCSD Pascal, program parameters are ignored; INPUT, OUTPUT, and KEYBOARD are always defined and opened. In Pascal-86, program parameters are implemented according to standard Pascal, with the addition of the preconnection facility. (4.2.1, 5.3.2)
- UCSD Pascal provides a dynamic string data type and several string manipulation procedures that operate on this type. In Pascal-86, strings are implemented according to standard Pascal; that is, as fixed-length packed arrays of characters. (5.3.2)
- In UCSD Pascal, the maximum size of a set is 255 words, or 4080 elements. In Pascal-86, the maximum size of a set is 32767 elements. (5.3.2)
- UCSD Pascal provides an INTERACTIVE file type for console input and output. Pascal-86 treats TEXT and FILE OF CHAR file types as interactive files. (5.3.2, 8.7)
- UCSD Pascal provides untyped files for device-dependent input and output. Pascal-86 provides typed files only, per standard Pascal. (5.3.2, 8.7)
- In UCSD Pascal, equality comparison between similar records and arrays is permitted. In Pascal-86, per standard Pascal, such comparisons are not permitted. (5.3.4)
- UCSD Pascal does not support procedural and functional parameters. Pascal-86 does support them, in accordance with standard Pascal. (6.4)
- If the case value on execution of a CASE statement does not match any of the case constants given, UCSD Pascal will not register an error. Pascal-86 does register an error unless an OTHERWISE clause (which may be empty) is present. (7.2.5)
- UCSD Pascal does not permit out-of-block GOTO statements. Pascal-86 does permit them, in accordance with the rules of standard Pascal. (7.2.10)
- UCSD Pascal provides an EXIT statement for exiting prematurely from a procedure or function. Pascal-86, in accordance with standard Pascal, does not provide such a statement.
- UCSD Pascal supports the built-in procedures MARK and RELEASE in place of DISPOSE, for deallocation of memory allotted to dynamic variables.Pascal-86 supports DISPOSE in accordance with standard Pascal. (8.5.2)
- UCSD Pascal does not support the built-in procedures PACK and UNPACK. Pascal-86 supports PACK and UNPACK in accordance with standard Pascal. (8.6)
- UCSD Pascal supports random access to files by means of the SEEK procedure and a position parameter to READ and WRITE. Pascal-86 supports sequential files only, in accordance with standard Pascal. (8.7)
- In UCSD Pascal, lazy I/O is used for operations on INTERACTIVE files. In Pascal-86, lazy I/O is used for operations on files of type TEXT and unpacked FILE OF CHAR. (8.7)
- In UCSD Pascal, all compiler controls within the source program are enclosed within comment brackets. In Pascal-86, controls are not enclosed within comment brackets, but appear on control lines beginning with a dollar sign in the leftmost column. (10)

Both UCSD Pascal and Pascal-86 extend RESET and REWRITE to take a second parameter, which supplies the external name for a pre-existing file. (8.7)

A.3 Areas Where Versions of Pascal Differ

Separate Compilation. Standard Pascal assumes a monolithic compilation structure, which is untenable on microcomputers and many minicomputers. A common extension is to permit an external procedure to be defined via an EXTERN directive. This directive may be used to communicate with separately compiled Pascal procedures or with modules written in other languages. Pascal-86 uses an interface specification, so that the definition of data and procedures is physically written down in only one place. Elaborate systems for creating the appropriate block structure environment to recompile a single Pascal procedure have also been devised.

Treatment of Program Parameters. The *Report* does not clearly state the exact function of program parameters, so naturally there is disagreement on this subject. Some compilers ignore program parameters entirely, whereas others rely on the program parameters as the sole link to the program environment.

Strings. Character strings are another weak point in the Pascal language. The UCSD Pascal STRING type, which is similar to a string in PL/I, is the most popular method of extension. Some purists insist that strings should be incorporated into the language as a file type. More elaborate implementations are usually avoided because of their overhead.

Size of Sets. The range of values in all data types varies from implementation to implementation, but the size of sets is usually the most restrictive. You can usually expect that SET OF CHAR will be supported; however, there are some notable cases in which it is not supported.

Type Compatibility. The rules for type compatibility are not rigorously defined by the *Report* or by the *User Manual.* Pascal-86, following the ISO Standard, enforces the strictest set of rules possible.

Procedural and Functional Parameters. Procedural and functional parameters are expensive to implement, and therefore are often omitted. Compilers also vary concerning which of the predefined functions and procedures, if any, may be passed as arguments.

Out-of-Block GOTO's. Out-of-block GOTO's are expensive to implement and seldom used, so they are frequently not supported.

MARK and RELEASE vs. DISPOSE. MARK and RELEASE are much easier to implement than DISPOSE, and so are favored by the smaller implementations for microcomputers and minicomputers.

Dynamic Association of Logical Files with Physical Files. The Pascal-86 extension to RESET and REWRITE is the most popular method for dynamic association of file variables with physical files.

Interactive I/O. The read-ahead feature of standard Pascal files does not work well with interactive devices—the console operator gets a prompt for input before the program can write instructions on how to answer the prompt. The favored solution to this problem is to use lazy input, which defers the action of a GET until the buffer variable is actually interrogated. UCSD Pascal uses lazy input only on files of the special INTERACTIVE type. In CDC implementations, extra syntax in the program heading identifies those files that are to be accessed using lazy input. Pascal-86 uses lazy input on all files declared as TEXT or FILE OF CHAR.

Compiler Controls. Since there is no mention of compiler controls in the *Report*, the *User Manual*, or the ISO Standard, it is not surprising that compilers vary both in the variety of controls offered and the method of specifying the controls. You should plan on changing all controls when you transport your Pascal programs to a new compiler.

APPENDIX B
PROCESSOR-DEPENDENT LANGUAGE FEATURES

The following Pascal-86 language features are provided especially for use in iAPX 86,88 microprocessor applications. After each listed feature is a reference to the chapter or section of this manual where the feature is described.

- Double (LONGREAL) and extended (TEMPREAL) precision for floating-point numbers, and extended precision for all real-type expressions and intermediate results (5.3.1, 7.1.4, 7.1.8)
- The Denormalized Operand exception in real arithmetic (7.1.8)
- The port input and output procedures INBYT, INWRD, OUTBYT, and OUTWRD (8.8)
- The interrupt control procedures SETINTERRUPT, ENABLEINTERRUPTS, DISABLEINTERRUPTS, and CAUSEINTERRUPT (8.9)
- The 8087 procedures GET8087ERRORS and MASK8087ERRORS (8.10)
- The predefined types AT87ERRORS and AT87EXCEPTIONS and the predefined constants AT87NVLD, AT87DENR, AT87ZDIV, AT87OVER, AT87UNDR, AT87PRCN, AT87RSVD, and AT87MASK (8.10)
- PQCLOSE describes a method for closing files in Pascal, Appendix B.
- RANDOM ACCESS I/O permits access to any file in Pascal, Appendix B.

B.1 PQCLOSE, Closing Files in Pascal

Standard Pascal does not provide a method to close a file. On systems such as Series III, which limit the number of files that may be open at one time, this might pose a problem. If you encounter a problem, the run-time system for Pascal provides a procedure to close a file.

CAUTION

To avoid deleting a file, read this entire section before executing PQCLOSE.

The PQCLOSE procedure is not a predefined procedure in Pascal; it must be declared in the interface specification of the module. The declaration should be as follows:

```
PUBLIC PQCLOSE;
      PROCEDURE PQCLOSE(VAR F : <file type>);
```

where

<file type> is the type of file you choose to close.

If you have more than one type of file that you choose to close in the same module, use the following declaration:

```
PUBLIC PQCLOSE;
      PROCEDURE PQCLOSE(VAR F : BYTES);
```

Files that have not been declared in the program heading or do not have a physical file name specified on the RESET or REWRITE are considered temporary files and will be deleted automatically either when closed or at program termination.

B.2 Random Access I/O

The six procedures and functions described in this section allow a Pascal program to access any record in a file without explicitly reading all records before the desired record is accessed. It is also possible to update a record without affecting any of the other records in the file.

The procedures (SETRANDOM, SEEKREAD, SEEKWRITE) and the functions (POSITION, ENDPOSITION, and EMPTY) that implement random I/O in Pascal are applicable to any file type except TEXT. To use the random access feature, you must declare the procedures SETRANDOM, SEEKREAD, SEEKWRITE, POSITION, ENDPOSITION, and EMPTY in the interface section of your program.

All of the procedures and functions use the standard Pascal calling sequence.

B.2.1 SETRANDOM

Call this procedure before the RESET or REWRITE on the file.

The purpose of SETRANDOM is to inform the run-time system that the file will be accessed randomly. As a result, subsequent RESET or REWRITE on the file will open the file for update rather than read-only or write-only mode.

After the RESET or REWRITE is performed, the file will be positioned at record number 0.

```
Procedure  SETRANDOM(var random file:BYTES);
```

where

random file is a Pascal file variable.

An exception will occur if a RESET or REWRITE already has been performed on the file unless it has been closed by PQCLOSE. Also, an exception will occur when the RESET or REWRITE is performed when the file is of type TEXT.

B.2.2 SEEKREAD

This procedure moves the current file pointer to the specified record in the file and sets the file mode to allow an input operation. The file's buffer variable is not changed. If the specified record number is beyond the end of the file, then the file will be positioned at the end, and the end of file flag will be set to true. The first record in a file is record number 0.

```
Procedure  SEEKREAD(var random-file:BYTES;
                        rec-num:LONGINT);
```

where

random-file is a file that has been opened for random access.

rec-num is a non-negative LONGINT value that specifies the record at which the file is to be positioned.

An exception occurs if *rec-num* is negative or if the file has not been opened for random access.

B.2.3 SEEKWRITE

This procedure moves the current file pointer to the specified record in the file and sets the file mode to allow an output operation. The file's buffer variable is not changed. If the specified record number is beyond the end of the file, an exception will occur.

```
Procedure SEEKWRITE(var random-file:BYTES
                  rec-num:LONGINT);
```

where

random-file is a file that has been opened for random access.

rec-num is a non-negative LONGINT value that specifies the record at which the file is to be positioned.

An exception will occur if *rec-num* is negative or if the file has not been opened for random access.

B.2.4 POSITION

This function returns the record number at which the file is currently positioned.

```
FUNCTION POSITION(var random-file:BYTES)LONGINT;
```

where

random-file is a file that has been opened for random access.

An exception will occur if the file is not opened for random access.

B.2.5 ENDPOSITION

This function returns the record number of the last record in the file.

```
FUNCTION ENDPOSITION(var random-file:BYTE):LONGINT
```

where

random-file is a file that has been opened for random access.

An exception will occur if the file is not opened for random access or if the file is empty.

B.2.6 EMPTY

This function returns TRUE if a random access file contains no records, FALSE otherwise.

```
FUNCTION EMPTY(var random-file):BOOLEAN;
```

where

random-file is a Pascal file variable that has been opened for random access.

An exception will occur if the file is not opened for random access.

APPENDIX C
COMPILER CAPACITY

Proponents of structured programming recommend making modules and program objects small so they are easier to understand and manage. Nevertheless, the Pascal-86 compiler can translate modules of considerable size, as indicated below. Following each listed item is a reference to the chapter or section of this manual where that language feature is described.

- An array that requires more than 65535 bytes of memory for data storage will be placed in more than one data segment. All other types of variables must fit into a single memory segment (65535 bytes) (4.2, 5.1, Appendix H).
- A constant or module may occupy up to 65535 bytes (4.2, 5.1, Appendix H).
- Program, module, and subsystem identifiers can be up to 31 characters long. All other PUBLIC identifiers can be up to 40 characters long (4.2).
- An index type for an array is an ordinal type whose values are a subset of the set of whole numbers. In Pascal-86 these values are four bytes long and lie within the range from −2,147,483,647 to +2,147,483,647.
- The number of characters in a string is limited only by memory size (5.3.2).
- A set may have a maximum of 32767 elements, and the ordinal values of set members must lie within the INTEGER range (5.3.2).
- An expression may have up to 100 operands (7.1).
- The ordinal value of a case constant in a CASE statement must be in the range from −32767 to 65535. Also, in any CASE statement, the difference between the largest and the smallest case constants must be less than 1009 (7.2.5).
- A string specifying a physical file must be in a form acceptable to the run-time operating system, and it can be up to 45 characters long (8.7.1, 8.7.2, 10.3.7, 10.3.10, 10.3.14, 10.3.15).
- The amount of memory designated for internal tables ranges from 5K to 64K (10.3.20).
- The total space occupied by the run-time stack may be up to 65535 bytes (Appendixes J and K).
- The dictionary summary provided in the listing will help to determine how the compiler is allocating memory for symbols in the system. In a 128K system, for example, the static symbol space is about 4K and more than about 90 symbols will cause the dynamic symbol table to spill to disk. In a 192K system, the static symbol space is 16K and the dynamic symbol table will spill after about 600 symbols.

APPENDIX D
LANGUAGE SYNTAX SUMMARY

This appendix presents the syntax of the Pascal-86 language, using the notation that appears in the text.

In this notation, the following conventions apply:

- Keywords, letter symbols, and punctuation symbols that you use verbatim in your programs—the *terminal symbols* of the language—are represented in monospace type, in which every character has the same width, just as it does in output media such as CRT console displays and printouts. All letters in terminal symbols are shown in upper case in the notation; however, you may use either upper case or lower case for these symbols in your programs. For example:

```
E        FOR        PROCEDURE
(        TO         TYPE
:=       DO         ARRAY
```

  are all terminal symbols.

- Terms standing for language elements or constructs that are defined elsewhere in this notation—in other words, *nonterminal symbols*—are represented in italicized lower-case letters in non-monospace type, in which the width of a character varies. For example:

  digits *variable*
  sign *expression*
  binary-digit *statement*

  are all nonterminal symbols.

- When two adjacent items must be concatenated, they appear with no space between them. A blank space between two items indicates that the two items may be separated by one or more *logical-blanks*. For example:

  digits . *digits* [E [*sign*]*digits*]

  specifies that the first set of *digits*, the . symbol, and the second set of *digits* must be concatenated, with no blanks between them. Likewise, the E symbol, the *sign* if included, and the third set of *digits* must be concatenated.

- Optional constructs are enclosed in square brackets set in light type, or in square brackets that are taller than a single line. (Square bracket symbols that are part of the Pascal-86 language are set in monospace type, a heavier type face, and are never taller than a single line of type.) For example, in the construct represented by:

  digits . *digits* [E [*sign*]*digits*]

  the first and second sets of *digits* and the . symbol are required, and the entire part following the second set of *digits* is optional. If this optional part is included, the *sign* may still be omitted.

- Optional constructs that can be repeated a number of times are marked by a three-dot ellipsis following the closing square bracket. For example:

  binary-digit [*binary-digit*] . . . B

  stands for a concatenated sequence of one or more *binary-digits* followed immediately by a **B** symbol.

- Alternative constructs are represented as vertically adjacent items separated by extra vertical spacing and enclosed between curly braces that are taller than a single line of type. When these braces appear, choose any one of the constructs enclosed between the braces. For example:

$$\left\{\begin{array}{l}\textit{digits}\\ \textit{binary-digit}\,[\textit{binary-digit}\,]\texttt{...B}\\ \textit{octal-digit}\,[\textit{octal-digit}\,]\texttt{...Q}\\ \textit{digit}\,[\textit{hex-digit}\,]\texttt{...H}\end{array}\right\}$$

  indicates that the construct described may have any one of the four forms listed between the large braces.

- Text enclosed between the character sequence (* and the sequence *), when these symbols are in light, non-monospace type, is a prose definition of the given construct. Such definitions are used when symbolic definitions would be more cumbersome. For example:

  (* any uppercase or lowercase letter of the alphabet *)

  is used to avoid listing 52 separate characters vertically between braces.

- The start of a new line in the notation does not mean you must start a new line at that point in your program; however, you may do so for readability. For example, when you use the construct:

```
FOR variable := expression TO expression
    DO statement
```

  you need not include a carriage return after the second *expression*, but in many programs doing so makes the statement more readable.

The gray-shaded portions of the notation denote features that are Intel extensions to standard Pascal.

D.1 Basic Alphabet and Tokens

letter (* any uppercase or lowercase letter of the alphabet *)

digit (* any single decimal digit *)

word-symbol (* any one of the following: AND ARRAY BEGIN CASE CONST DIV DO DOWNTO ELSE END FILE FOR FUNCTION GOTO IF IN LABEL MOD MODULE NIL NOT OF OR OTHERWISE PACKED PRIVATE PROCEDURE PROGRAM PUBLIC RECORD REPEAT SET THEN TO TYPE UNTIL VAR WHILE WITH *)

special-symbol (* any one of the following: + - * / = <> < > <= >= ( ) [ ] { } := . , ; : ' ↑ .. *) @

input $\left[\left\{\begin{array}{l}\textit{token}\\ \textit{logical-blank}\end{array}\right\}\right]\ldots$

token	(* any one of the following: *identifier*, *integer*, *real-number*, *string*, *special-symbol* *)
logical-blank	(* any one of the following: blank character, carriage return character, line feed character, tab character, *comment* *)
comment	$\left\{\begin{array}{c}(*\\ \{\end{array}\right\}$ [*non-closure*]... $\left\{\begin{array}{c}*)\\ \}\end{array}\right\}$
non-closure	(* any character except (1) a Zright brace, or (2) a star that is immediately followed by a right parenthesis *)
identifier	$\mathit{letter}\left[\left\{\begin{array}{c}\mathit{letter}\\ \_\\ \mathit{digit}\end{array}\right\}\right]\ldots$
signed-integer	[*sign*]*integer*
sign	(* either a plus or a minus sign *)
integer	$\left\{\begin{array}{l}\mathit{digits}\\ \mathit{binary\text{-}digit}[\mathit{binary\text{-}digit}]\ldots\ \mathtt{B}\\ \mathit{octal\text{-}digit}[\mathit{octal\text{-}digit}]\ldots\ \mathtt{Q}\\ \mathit{digit}[\mathit{hex\text{-}digit}]\ldots\ \mathtt{H}\end{array}\right\}$
digits	*digit* [*digit*]...
binary-digit	(* either a 0 or a 1 *)
octal-digit	(* any single digit from 0 through 7 *)
hex-digit	(* any single decimal digit or any upper- or lower-case letter from A through F *)
signed-real-number	[*sign*] *real-number*
real-number	$\left\{\begin{array}{l}\mathit{digits}\,.\,\mathit{digits}\ \ [\,\mathtt{E}\,[\mathit{sign}]\mathit{digits}]\\ \mathit{digits}\,\mathtt{E}\,[\mathit{sign}]\mathit{digits}\end{array}\right\}$
label	*digits*
string	*' character* [*character*]... *'* [*logical-blank ' character* [*character*]... *'*]...
character	$\left\{\begin{array}{l}\text{' '}\\ \text{(* any single printable ASCII character other than}\\ \text{an apostrophe, carriage return, or line feed *)}\end{array}\right\}$

D.2 Modularization and Block Structure

compilation	{ main-module non-main-module }
main-module	[module-heading ; [interface-spec]] program-heading ; block .
non-main-module	[module-heading ; [interface-spec]] private-heading ; declordefn .
module-heading	MODULE identifier
interface-spec	PUBLIC identifier ; section [PUBLIC identifier ; section]...
section	subsection [subsection]...
subsection	[FOR identifier [, identifier]... ;] [label-decl] [CONST constant-defn ; [constant-defn ;]...] [TYPE type-defn ; [type-defn ;]...] [VAR variable-decl ; [variable-decl ;]...] [prochdg-or-funchdg ;]...
program-heading	PROGRAM identifier [(prog-parameter-list)]
block	[label-decl] [CONST constant-defn ; [constant-defn ;]...] [TYPE type-defn ; [type-defn ;]...] [VAR variable-decl ; [variable-decl ;]...] [proc-or-func ;]... statement-part
private-heading	PRIVATE identifier
declordefn	[CONST constant-defn ; [constant-defn ;]...] [TYPE type-defn ; [type-defn ;]...] [VAR variable-decl ; [variable-decl ;]...] [proc-or-func ;]...
label-decl	LABEL label [, label]... ;
proc-or-func	{ procedure-decl function-decl }
prochdg-or-funchdg	{ procedure-heading function-heading }
statement-part	compound-statement

D.3 Constants, Types, and Variables

constant-defn	*identifier* = *constant*
constant	*string* *signed-integer* *signed-real-number* *constant-id* *sign numeric-constant-id*
constant-id	(* an *identifier* that stands for any *constant* *)
numeric-constant-id	(* an *identifier* that stands for a *signed-integer* or a *signed-real-number* *)
type-defn	*identifier* = *type-spec*
type-spec	*type-id* *enumerated-type* *subrange-type* *pointer-type* [PACKED] *array-type* [PACKED] *record-type* [PACKED] *set-type* [PACKED] *file-type*
type-id	(* an *identifier* that stands for a type *)
enumerated-type	(*identifier* [, *identifier*]...)
subrange-type	*ordinal-constant* . . *ordinal-constant*
ordinal-constant	(* a constant of an *ordinal-type* *)
ordinal-type	*enumerated-type* *subrange-type* (* a *type-id* that stands for an *enumerated-type*, a *subrange-type*, or one of the predefined types INTEGER, WORD, LONGINT, BOOLEAN, or CHAR *)
pointer-type	↑ *type-id* or @ *type-id*
array-type	ARRAY [*index-type* [, *index-type*]...] OF *component-type*
index-type	*ordinal-type*
component-type	*type-spec*
record-type	RECORD [*field-list* [;]] END

field-list	*field-id* [, *field-id*]... : *type-spec* [; *field-id* [, *field-id*]... : *type-spec*]... [*field-id* [, *field-id*]... : *type-spec* [; *field-id* [, *field-id*]... : *type-spec*]...] CASE [*field-id* :] *tagtype-id* OF *case-const* [, *case-const*]... : ([*field-list* [;]]) [; *case-const* [, *case-const*]... : ([*field-list* [;]])]...
field-id	*identifier*
tagtype-id	(* a *type-id* that stands for an *ordinal-type* *)
case-const	*ordinal-constant*
set-type	SET OF *ordinal-type*
file-type	FILE OF *type-spec*
variable-decl	*identifier* [, *identifier*]... : *type-spec*
variable	*entire-variable* *indexed-variable* *field-designator* *buffer-variable* *referenced-variable*
entire-variable	(* an *identifier* that stands for a variable *)
indexed-variable	*array-variable* [*expression* [, *expression*]...]
field-designator	*record-variable* . *field-id*
buffer-variable	*file-variable* ↑ or *file-variable* @
referenced-variable	*pointer-id* ↑ or *pointer-id* @
array-variable	(* a *variable* of an array type *)
record-variable	(* a *variable* of a record type *)
file-variable	(* a *variable* of a file type *)
pointer-variable	(* a *variable* of a pointer type *)

D.4 Procedures and Functions

procedure-decl	*procedure-heading* ; *block* *procedure-heading* ; FORWARD
procedure-heading	PROCEDURE *identifier* [(*parameter-list*)]

```
function-decl           { function-heading ;
                          block

                          FUNCTION identifier ;
                          block

                          function-heading ;
                          FORWARD }

function-heading        FUNCTION identifier [( parameter-list )] : type-id ;

parameter-list          parameters [; parameters]...

parameters                identifier [, identifier]... : type-id

                        { VAR identifier [, identifier]... : type-id
                          VAR identifier [, identifier].. : BYTES

                          PROCEDURE identifier [, identifier]...
                              [( parameter-list )]

                          FUNCTION identifier [, identifier]
                              [( parameter-list )] : type-id }
```

D.5 Expressions and Statements

```
expression              simple-expression [relational-op simple-expression]

simple-expression       [sign] term [adding-op term]...

term                    factor [multiplying-op factor]...

factor                  { variable
                          constant-id
                          integer
                          real-number
                          string
                          function-designator
                          NIL
                          ( expression )
                          NOT factor
                          [ element [, element]... ]
                          string-constant-id [ expression ] }

relational-op           (* any one of the following: = <> < > <= >= IN *)

adding-op               (* any one of the following: + - OR *)

multiplying-op          (* any one of the following: * / DIV MOD AND *)

function-designator     function-id [( argument [, argument]... )]

argument                { expression [: expression [: expression]]
                          procedure-id }
```

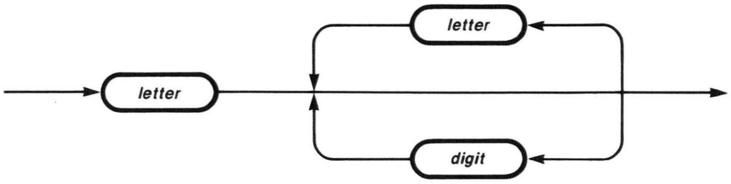
function-id	(* an *identifier* that stands for a function *)
procedure-id	(* an *identifier* that stands for a procedure *)
element	*expression* [. . *expression*]
string-constant-id	(* an *identifier* that stands for a string constant *)
statement	[*label* :] { (* null, or empty, statement *) *assignment-statement* *procedure-statement* *compound-statement* *IF-statement* *CASE-statement* *WHILE-statement* *REPEAT-statement* *FOR-statement* *WITH-statement* *GOTO-statement* }
assignment-statement	*variable* := *expression*
procedure-statement	*procedure-id* [(*argument* [, *argument*]...)]
compound-statement	BEGIN *statement* [; *statement*]... END
IF-statement	IF *expression* THEN *statement* [ELSE *statement*]
CASE-statement	{ CASE *expression* OF [*case-const* [, *case-const*]... : *statement* ;]... *case-const* [, *case-const*]... : *statement* [;] END CASE *expression* OF [*case-const* [, *case-const*]... : *statement* ;]... OTHERWISE *statement* [; *statement*]... END }
WHILE-statement	WHILE *expression* DO *statement*
REPEAT-statement	REPEAT *statement* [; *statement*]... UNTIL *expression*
FOR-statement	{ FOR *variable* := *expression* TO *expression* DO *statement* FOR *variable* := *expression* DOWNTO *expression* DO *statement* }
WITH-statement	WITH *variable* [, *variable*]... DO *statement*
GOTO-statement	GOTO *label*

APPENDIX E
SYNTAX DIAGRAMS

This appendix presents the syntax of the Pascal-86 language in the form of syntax diagrams, as used in an appendix to Jensen and Wirth's *Pascal User Manual* and in a number of textbooks on Pascal.

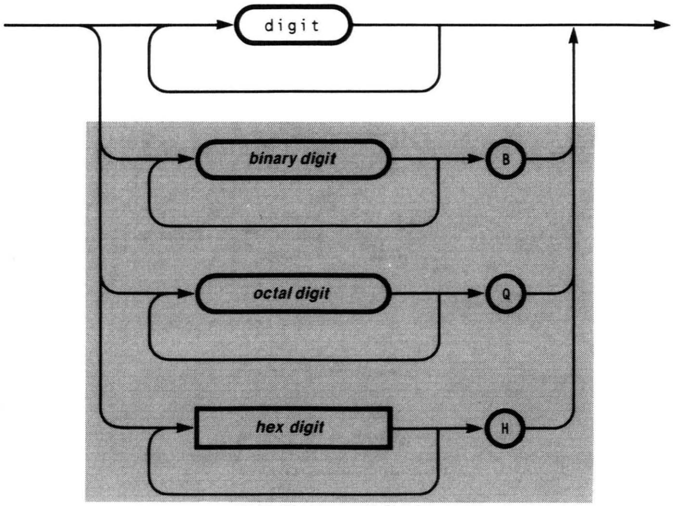
In these diagrams, every path you can follow, going in the direction of the arrows, represents a syntactically correct construct in Pascal-86. Keywords, letter symbols, and punctuation symbols that you use verbatim in your programs—the *terminal symbols* of the language—are enclosed in circles or ovals. Terms standing for language constructs that are defined in their own syntax diagrams—in other words, *nonterminal symbols*—are enclosed in rectangular boxes.

The gray-shaded portions of the diagrams denote features that are Intel extensions to standard Pascal.



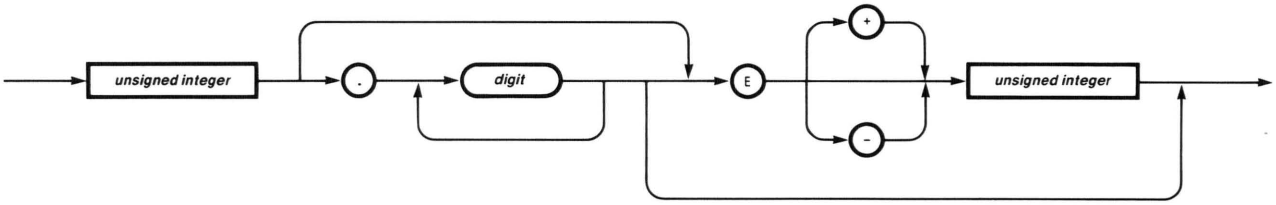
identifier

121539-1



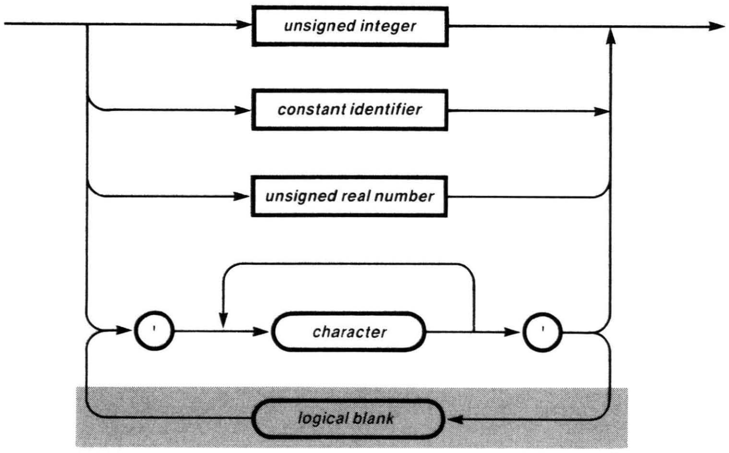
unsigned integer

121539-2



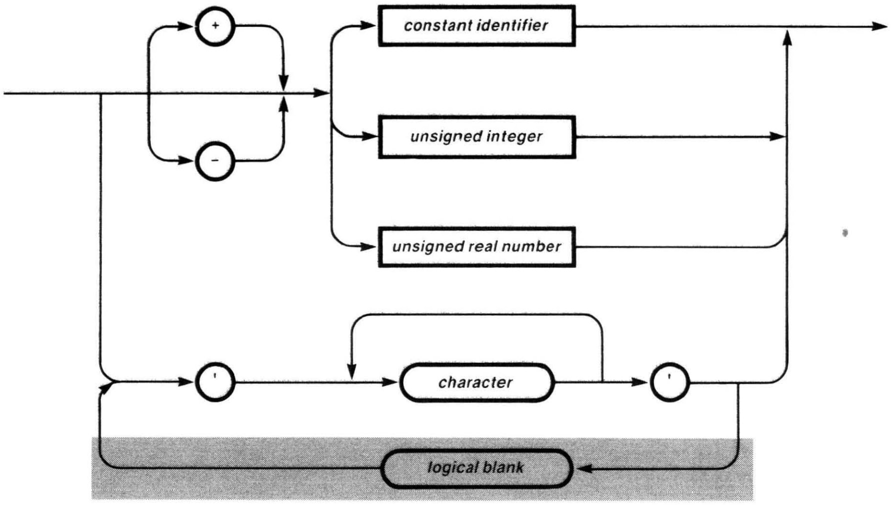
unsigned real number

121539-3



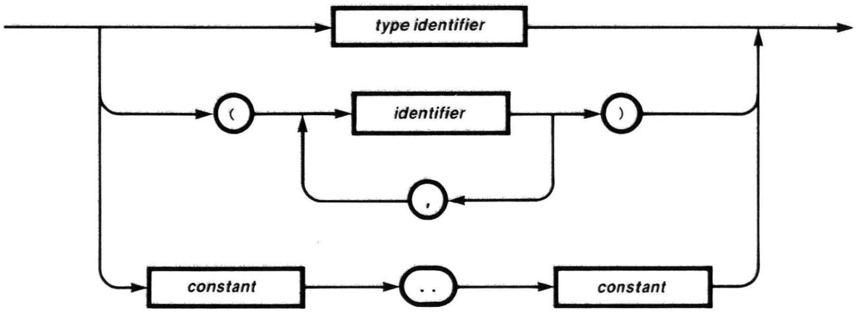
unsigned constant

121539-4



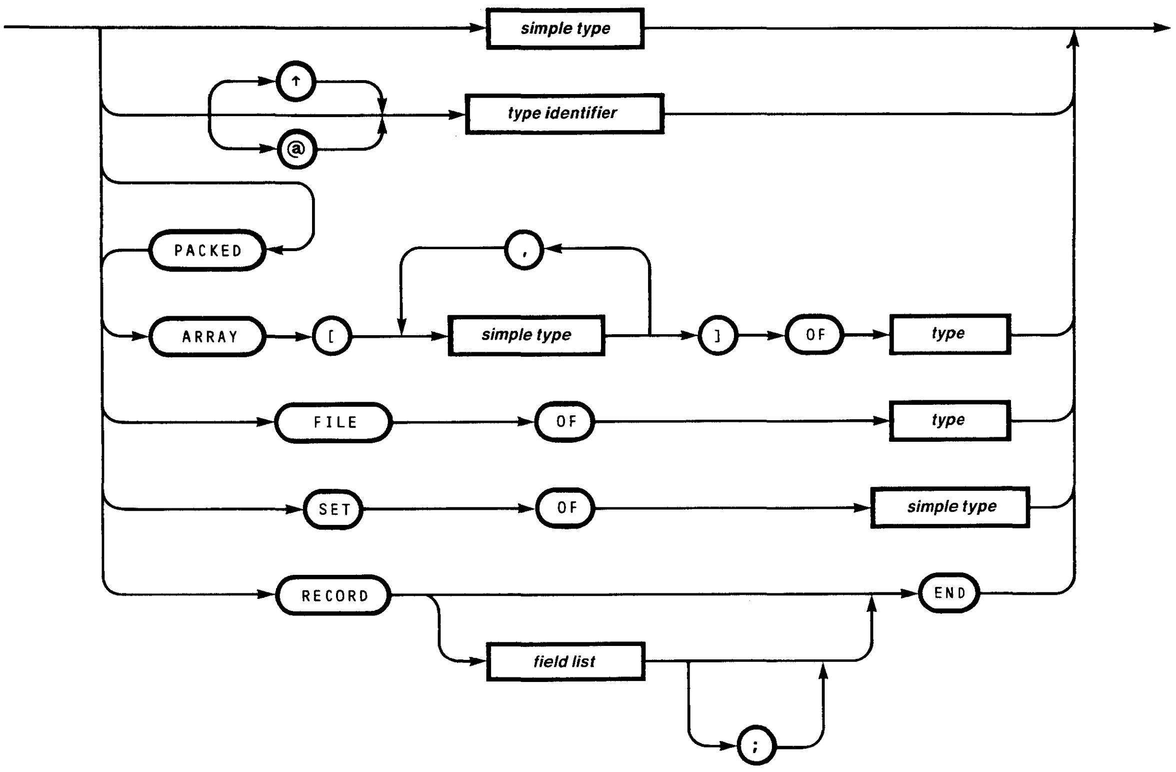
constant

121539-5



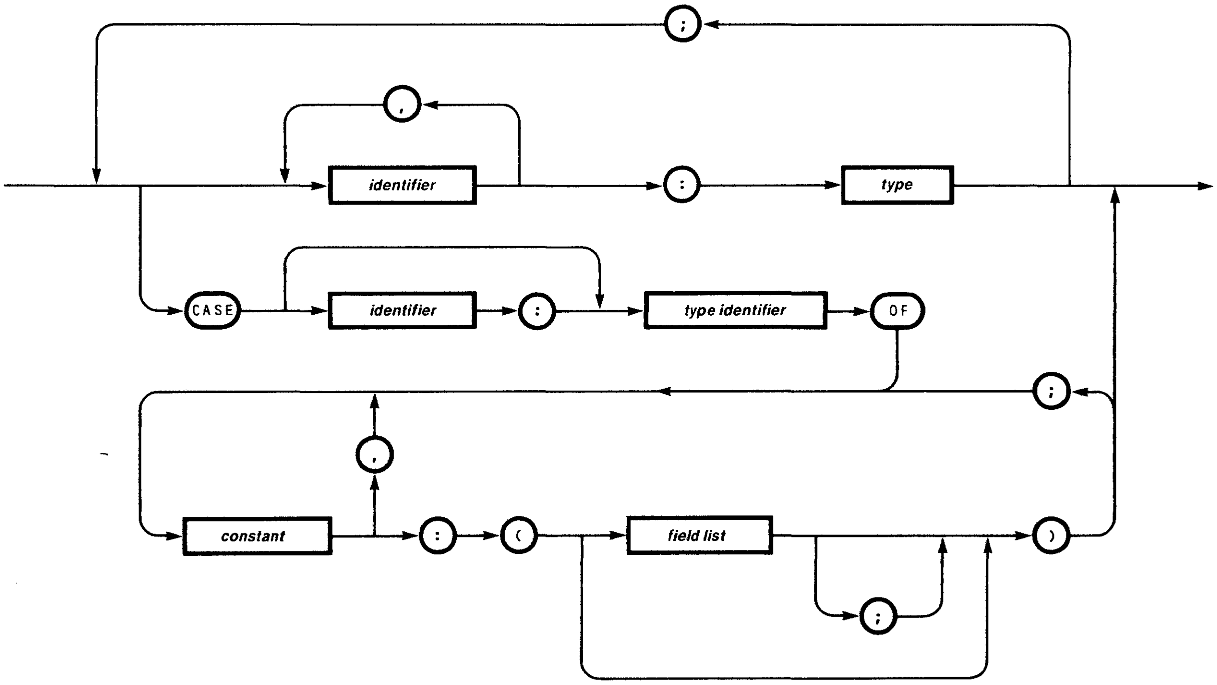
simple type

121539-20



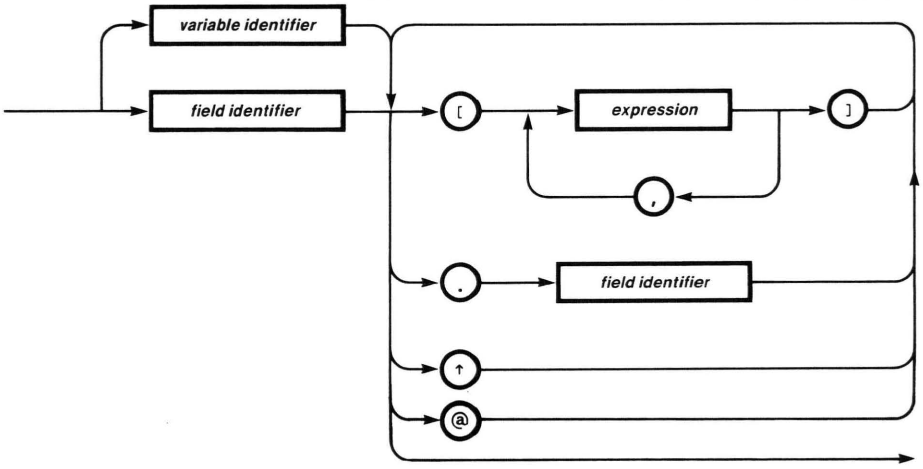
type

121539-6



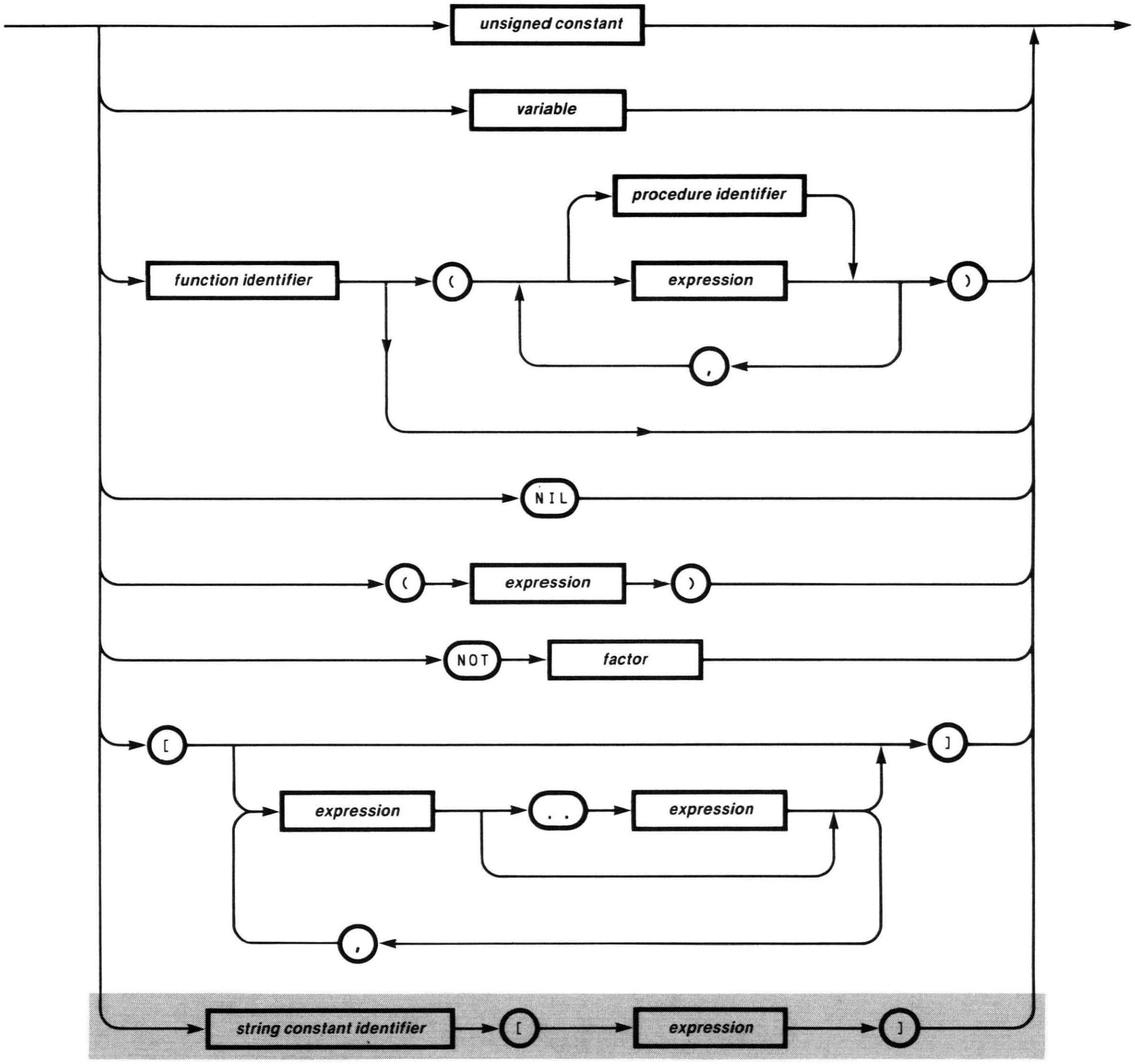
field list

121539-7



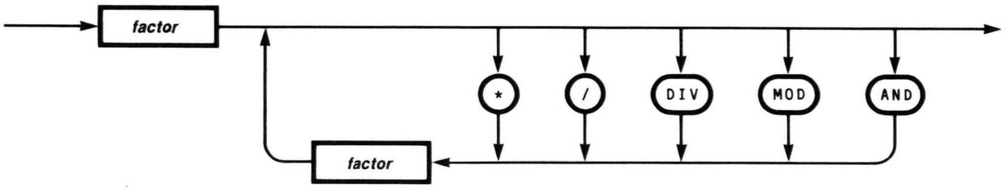
variable

121539-8



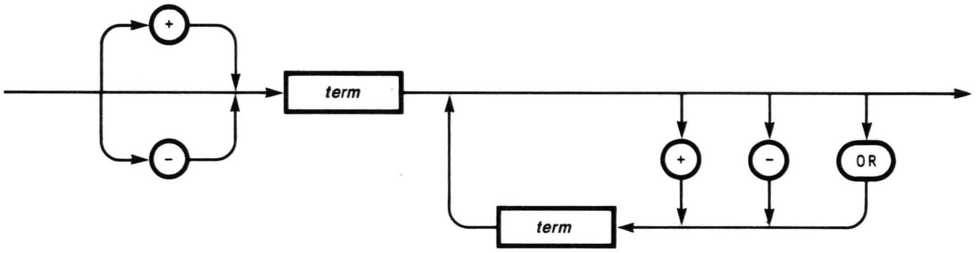
factor

121539-9



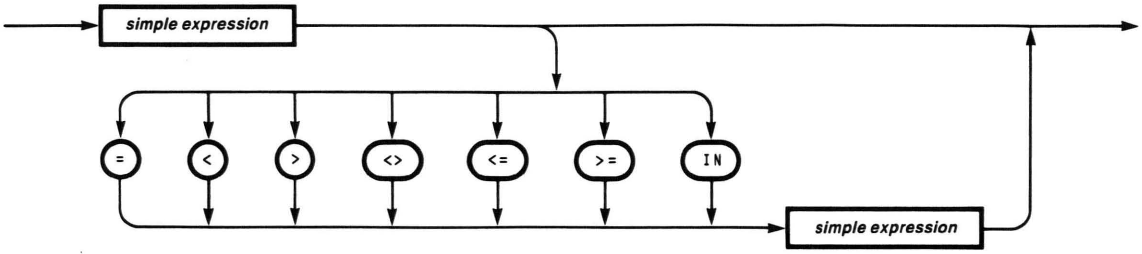
term

121539-10



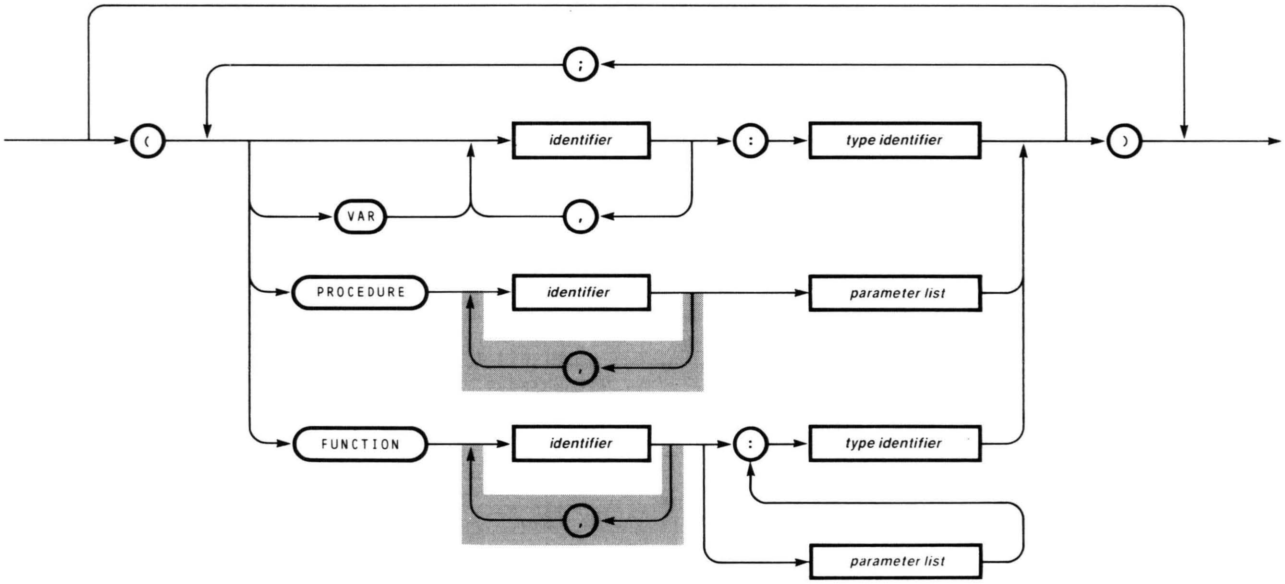
simple expression

121539-11



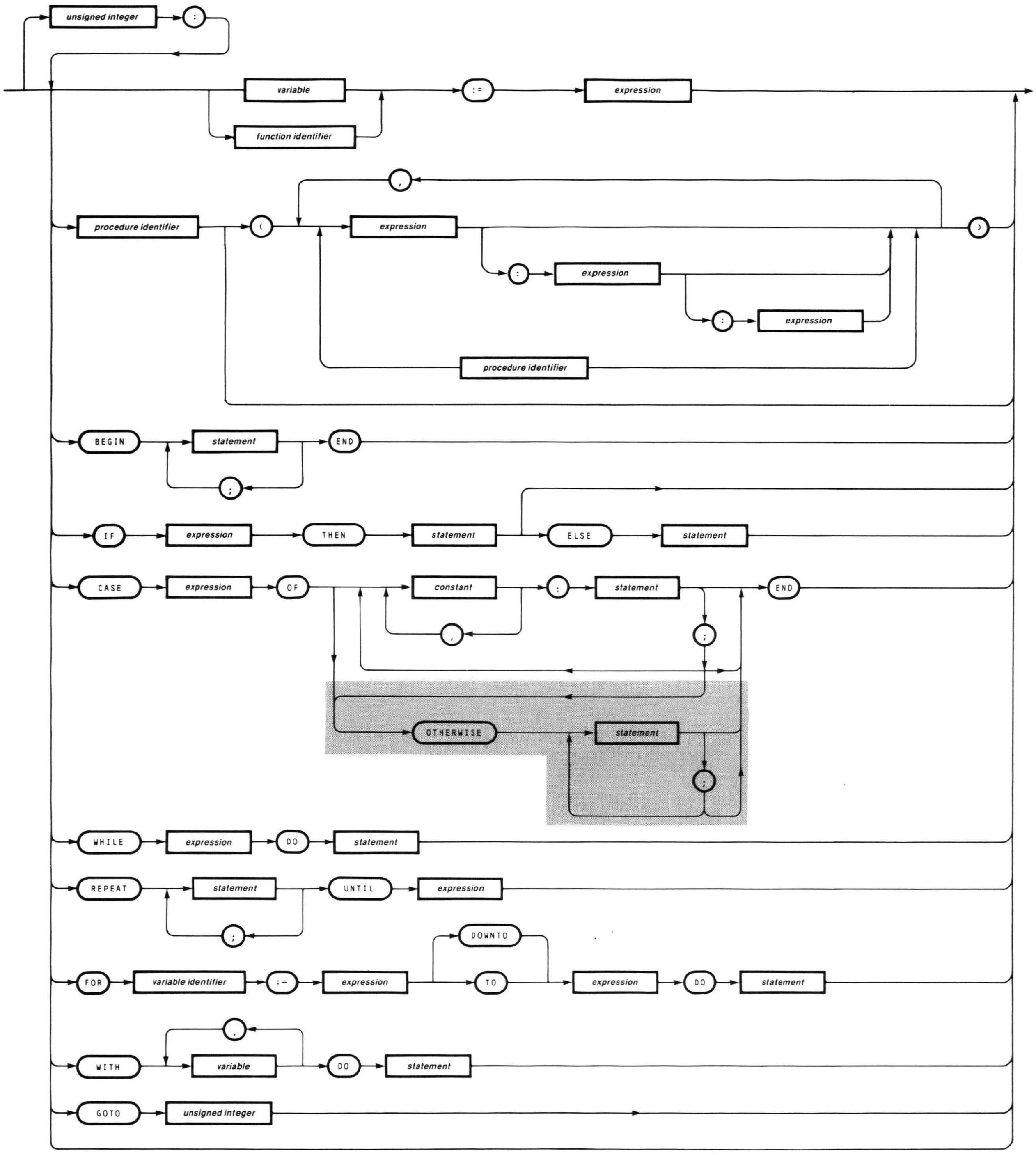
expression

121539-12



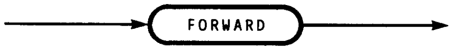
parameter list

121539-13



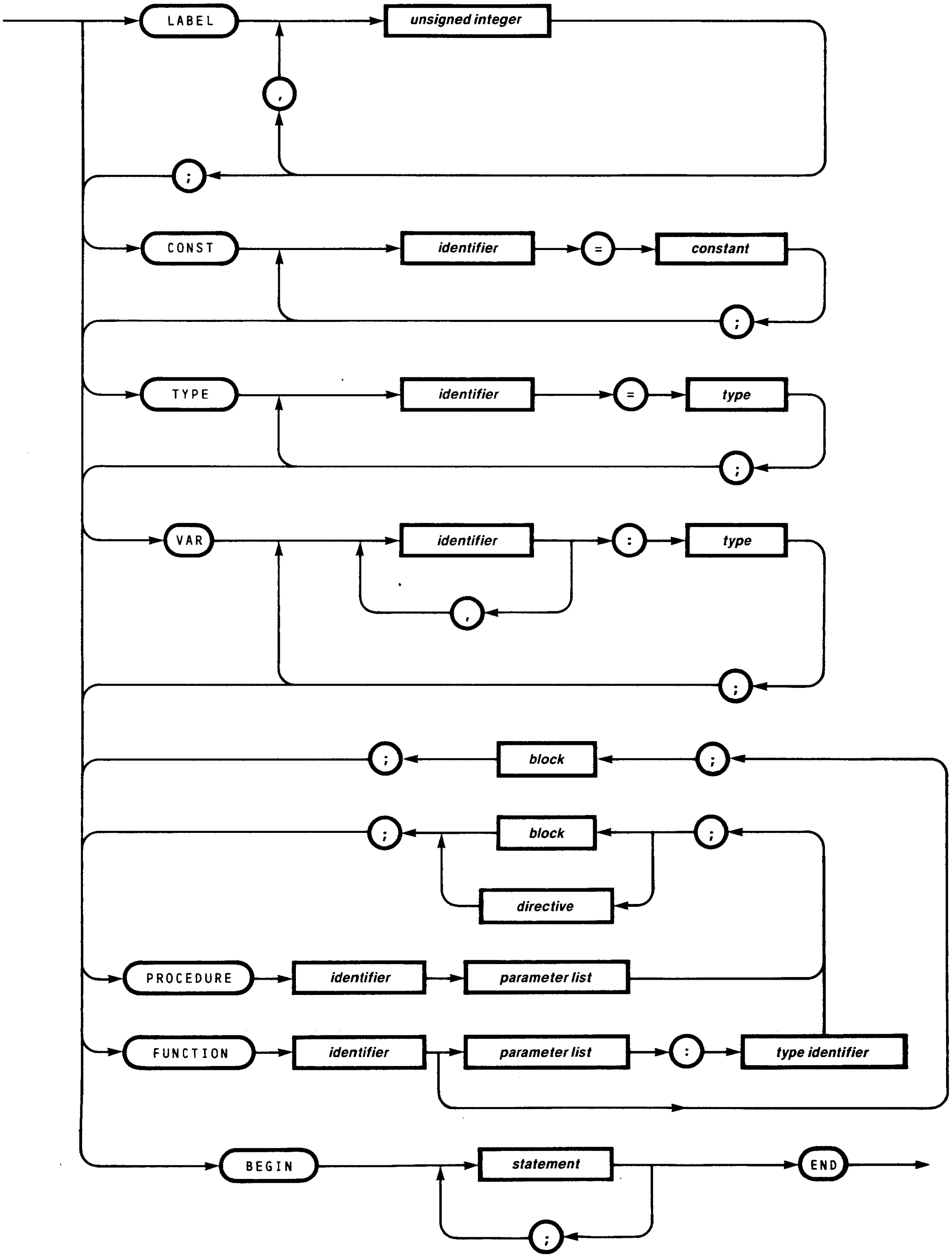
statement

121539-14



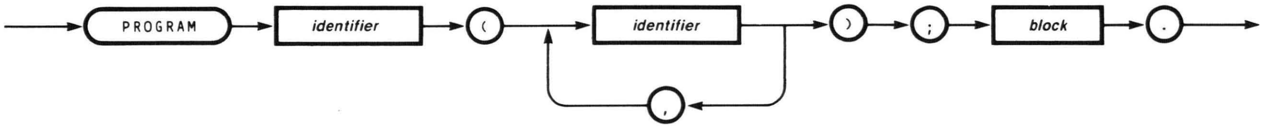
directive

121539-21



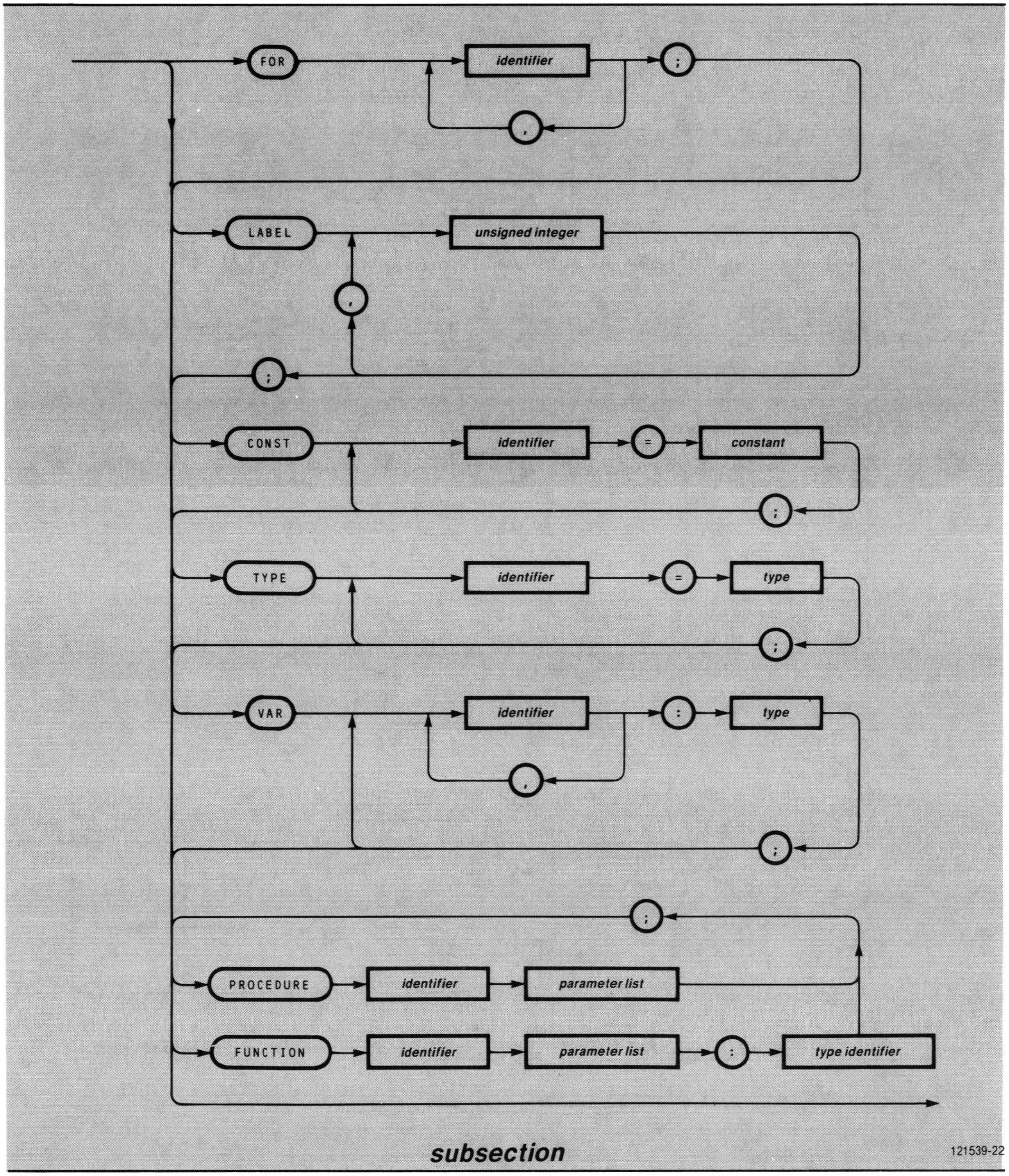
block

121539-15



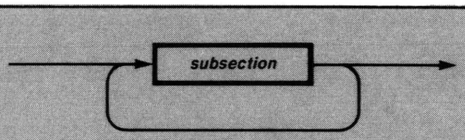
program

121539-16



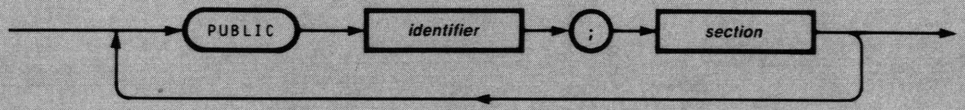
subsection

121539-22



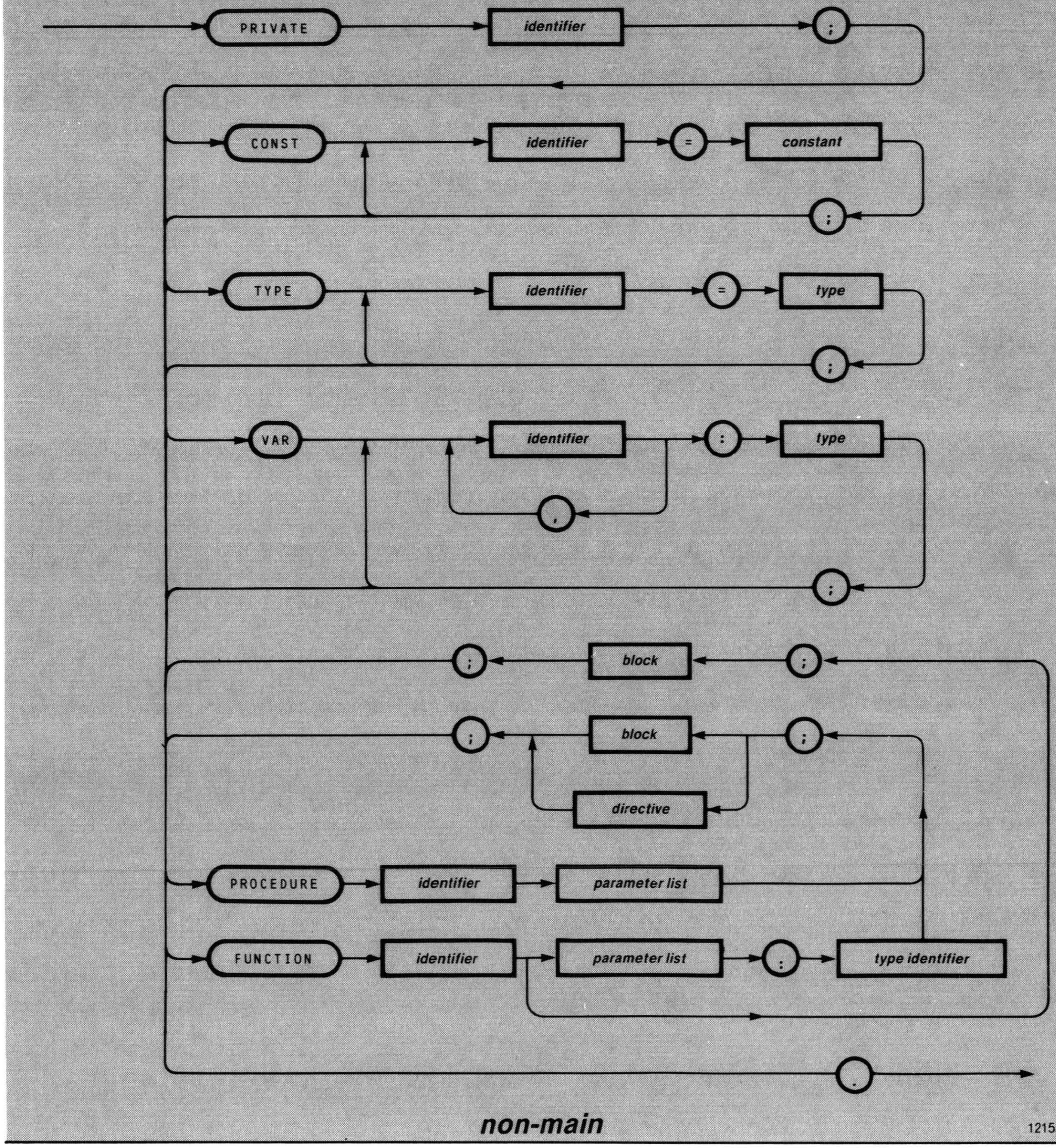
section

121539-23



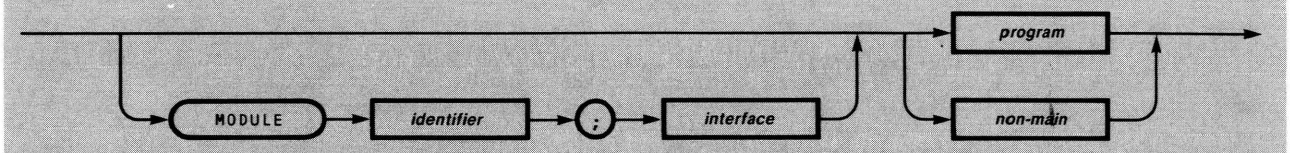
interface

121539-17



non-main

121539-18



module

121539-19

APPENDIX F
PASCAL-86 VOCABULARY

Tables F-1 through F-8 summarize all keywords, special punctuation symbols, directives, and predefined identifiers in Pascal-86, giving a brief definition or classification of each. The gray-shaded portions describe features that are Intel extensions.

Table F-1. Keywords

Keyword	Meaning
AND	Boolean operator
ARRAY	Type constructor
BEGIN	Start of compound statement
CASE	Start of CASE statement or record variant
CONST	Start of constant definition
DIV	Arithmetic operator
DO	Part of WHILE, FOR, or WITH statement
DOWNTO	Part of FOR statement, decrementing form
ELSE	Part of IF statement
END	End of record type specification, compound statement, or CASE statement
FILE	Type constructor
FOR	Start of FOR statement, or start of recipients clause in interface specification for separate compilation
FUNCTION	Start of functionn declaration
GOTO	Start of GOTO statement
IF	Start of IF statement
IN	Relational operator (for sets)
LABEL	Start of label declaration
MOD	Arithmetic operator
MODULE	Start of module heading for separate compilation
NIL	Null value for pointer variable
NOT	Boolean operator
OF	Part of array type specification, record variant, or CASE statement
OR	Boolean operator
OTHERWISE	Start of optional clause in CASE statement
PACKED	Optional prefix to structured type specification
PRIVATE	Start of private heading of separate compilation
PROCEDURE	Start of procedure declaration
PROGRAM	Start of program heading
PUBLIC	Lead-in to public section in interface specification for separate compilation
RECORD	Type constructor
REPEAT	Start of REPEAT statement
SET	Type constructor
THEN	Part of IF statement
TO	Part of FOR statement, incrementing form
TYPE	Start of type definition
UNTIL	Part of REPEAT statement
VAR	Start of variable declaration
WHILE	Start of WHILE statement
WITH	Start of WITH statement

Table F-2. Special Symbols

Symbol	Description	Meaning
+	plus sign	arithmetic or set operator, or sign part of numeric constant
−	minus sign	arithmetic or set operator, or sign part of numeric constant
*	star	arithmetic or set operator, or part of comment bracketing symbol
/	slash	arithmetic operator
=	equal sign	relational operator, or part of constant or type definition
<>	"not equal" symbol	relational operator
<	"less than" symbol	relational operator
>	"greater than" symbol	relational operator
≤	"less than or equal to" symbol	relational operator
≥	"greater than or equal to" symbol	relational operator
(	left parenthesis	bracketing symbol for expression factor, enumerated type specification, case constant list, parameter list, argument list, or program parameter list; or part of comment bracketing symbol
)	right parenthesis	bracketing symbol for expression factor, enumerated type specification, case constant list, parameter list, argument list, or program parameter list; or part of comment bracketing symbol
[	left bracket	bracketing symbol in array type specification, indexed variable, or set expression
]	right bracket	bracketing symbol in array type specification, indexed variable, or set expression
{	left brace	comment bracketing symbol
}	right brace	comment bracketing symbol
:=	assignment symbol	part of assignment statement or FOR statement
.	period, or dot	end of compilation, part of field designator, or decimal point in signed real number
,	comma	separator for small syntactic units in a sequence, such as labels or identifiers
;	semicolon	separator for large syntactic units in a sequence, such as declarations, definitions, and statements
:	colon	part of variable declaration, record, type specification, function declaration, functional parameter, parameter list, labeled statement, or CASE statement
'	apostrophe	bracketing symbol for literal string
↑	up-arrow	part of pointer type specification, referenced variable, or buffer variable
@	"at" sign	part of pointer type specification, referenced variable, or buffer variable
..	ellipsis	part of subrange type specification, or part of element in set expression
_	underscore	part of an identifier

Table F-3. Directives

Directive	Meaning
FORWARD	designates that the body of the named procedure or function is given later in this compilation

Table F-4. Predefined Program Parameters

Identifier	Meaning
INPUT	standard input file; of type TEXT
OUTPUT	standard output file; of type TEXT

Table F-5. Predefined Types

Identifier	Meaning
AT87ERRORS	SET OF AT87EXCEPTIONS
AT87EXCEPTIONS	Enumerated type that can be any of the 8087 exception constants listed in F.6
BOOLEAN	Simple ordinal type
BYTES	Variable parameter type used to communicate with external procedures
CHAR	Simple ordinal type
INTEGER	Simple ordinal type
WORD	Simple ordinal type
LONGINT	Simple ordinal type
LONGREAL	Simple real type
REAL	Simple real type
TEMPREAL	Simple real type
TEXT	FILE OF CHAR with line markers

Table F-6. Predefined Constants

Identifier	Meaning
AT87DENR	8087 denormalized operation exception bit
AT87MASK	8087 interrupt enable mask bit
AT87NVLD	8087 invalid operation exception bit
AT87OVER	8087 overflow exception bit
AT87PRCN	8087 precision exception bit
AT87RSVD	8087 reserved exception bit
AT87UNDR	8087 underflow exception bit
AT87ZDIV	8087 zero divide exception bit
CR	ASCII carriage return character of type CHAR
FALSE	Boolean constant
LF	ASCII line feed character of type CHAR
MAXINT	Maximum legal INTEGER value (32767 in Pascal-86)
MAXLONGINT	Maximum legal LONGINT value (2147483647 in Pascal-86)
MAXWORD	Maximum legal WORD value (65535 in Pascal-86)
TRUE	Boolean constant

Table F-7. Predefined Functions

Identifier	Parameter List*	Returned Value	Operation
ABS	(*arith-expr*)	integer or real	returns absolute value of argument
ARCCOS	(*arith-expr*)	real	returns arccosine of argument
ARCSIN	(*arith-expr*)	real	returns arcsine of argument
ARCTAN	(*arith-expr*)	real	returns arctangent of argument
CHR	(*int-expr*)	CHAR	returns corresponding character value
COS	(*arith-expr*)	real	returns cosine of argument
EOF	[(*file-var*)]	BOOLEAN	checks for end of file
EOLN	[(*textfile-var*)]	BOOLEAN	checks for end of line
EXP	(*arith-expr*)	real	returns e^x, where x is argument
LN	(*arith-expr*)	real	returns natural logarithm of argument
LORD	(*ord-expr*)	LONGINT	returns ordinal number of argument
LROUND	(*real-expr*)	LONGINT	returns rounded value of argument
LTRUNC	(*real-expr*)	LONGINT	returns integer part of argument
ODD	(*int-expr*)	BOOLEAN	checks whether argument is odd
ORD	(*ord-expr*)	INTEGER	returns ordinal number of argument
PRED	(*ord-expr*)	ordinal	returns preceding value in sequence
ROUND	(*real-expr*)	INTEGER	returns rounded value of argument
SIN	(*arith-expr*)	real	returns sine of argument
SQR	(*arith-expr*)	integer or real	returns square of argument
SQRT	(*arith-expr*)	real	returns square root of argument
SUCC	(*ord-expr*)	ordinal	returns next value in sequence
TAN	(*arith-expr*)	real	returns tangent of argument
TRUNC	(*real-expr*)	INTEGER	returns integer part of argument
WRD	(*ord-expr*)	WORD	returns ordinal number of argument

*For readability, optional blanks have been omitted from the syntax.

Table F-8. Predefined Procedures

Identifier	Parameter List*	Operation
CAUSEINTERRUPT	(*int-expr*)	causes specified 8086/8088 interrupt
DISABLEINTERRUPTS	none	disables 8086/8088 interrupts
DISPOSE	(*pointer*[,*tag*]...)	frees a dynamic variable
ENABLEINTERRUPTS	none	enables 8086/8088 interrupts
GET	[(*file-var*)]	inputs one file component
GET8087ERRORS	(*variable*)	returns 8087 error field, clears exceptions
INBYT	(*port-address,variable*)	inputs one byte from an 8086/8088 port
INWRD	(*port-address,variable*)	inputs two bytes from an 8086/8088 port
MASK8087ERRORS	(*expression*)	sets 8087 error mask
NEW	(*pointer*[,*tag*]...)	allocates a new dynamic variable
OUTBYT	(*port-address,expression*)	outputs one byte to an 8086/8088 port
OUTWRD	(*port-address,expression*)	outputs two bytes to an 8086/8088 port
PACK	(*unpacked-array, ord-expr, packed-array*)	packs an array
PAGE	[(*textfile-var*)]	causes page eject in text file output

Table F-8. Predefined Procedures (Cont'd.)

Identifier	Parameter List*	Operation
PUT	[(*file-var*)]	outputs one file component
READ	([*file-var*,]*variable* [,*variable*]...)	inputs one or more file components to variables
READLN	([*textfile-var*,]*variable* [,*variable*]...) or (*textfile-var*) or none	inputs zero or more text file components to variables and skips to next line
RESET	(*file-var*[,*string-expr*])	prepares a file for reading
REWRITE	(*file-var*[,*string-expr*])	prepares a file for writing omitted from the syntax.
SETINTERRUPT	(*int-expr,identifier*)	associated named interrupt procedure with given interrupt number
UNPACK	(*packed-array*, *unpacked-array*,*ord-expr*)	unpacks an array
WRITE	([*file-var*,]*write-param* [,*write-param*]...)	outputs one or more file components from variables
WRITELN	([*textfile-var*,]*write-param* [,*write-param*]...) or (*textfile-var*) or none	outputs zero or more text file components from variables and starts new line

*For readability, optional blanks have been omitted from the syntax.

APPENDIX G
CHARACTER SETS AND COLLATING SEQUENCE

ASCII CHARACTER	HEX	PASCAL-86 CHARACTER?	ASCII CHARACTER	HEX	PASCAL-86 CHARACTER?
NUL	00	no	@	40	yes
SOH	01	no	A	41	yes
STX	02	no	B	42	yes
ETX	03	no	C	43	yes
EOT	04	no	D	44	yes
ENQ	05	no	E	45	yes
ACK	06	no	F	46	yes
BEL	07	no	G	47	yes
BS	08	no	H	48	yes
HT	09	no	I	49	yes
LF	0A	no	J	4A	yes
VT	0B	no	K	4B	yes
FF	0C	no	L	4C	yes
CR	0D	no	M	4D	yes
SO	0E	no	N	4E	yes
SI	0F	no	O	4F	yes
DLE	10	no	P	50	yes
DCI	11	no	Q	51	yes
DC2	12	no	R	52	yes
DC3	13	no	S	53	yes
DC4	14	no	T	54	yes
NAK	15	no	U	55	yes
SYN	16	no	V	56	yes
ETB	17	no	W	57	yes
CAN	18	no	X	58	yes
EM	19	no	Y	59	yes
SUB	1A	no	Z	5A	yes
ESC	1B	no	[	5B	yes
FS	1C	no	\	5C	no
GS	1D	no	]	5D	yes
RS	1E	no	^(↑)	5E	yes
US	1F	no	—	5F	yes
space	20	yes	`	60	no
!	21	no	a	61	yes
"	22	no	b	62	yes
#	23	no	c	63	yes
$	24	no	d	64	yes
%	25	no	e	65	yes
&	26	no	f	66	yes
'	27	yes	g	67	yes
(	28	yes	h	68	yes
)	29	yes	i	69	yes
*	2A	yes	j	6A	yes
+	2B	yes	k	6B	yes
,	2C	yes	l	6C	yes
—	2D	yes	m	6D	yes
.	2E	yes	n	6E	yes
/	2F	yes	o	6F	yes
0	30	yes	p	70	yes
1	31	yes	q	71	yes
2	32	yes	r	72	yes
3	33	yes	s	73	yes
4	34	yes	t	74	yes
5	35	yes	u	75	yes
6	36	yes	v	76	yes
7	37	yes	w	77	yes
8	38	yes	x	78	yes
9	39	yes	y	79	yes
:	3A	yes	z	7A	yes
;	3B	yes	{	7B	yes
<	3C	yes	|	7C	no
=	3D	yes	}	7D	yes
>	3E	yes	~	7E	no
?	3F	no	DEL	7F	no

APPENDIX H RUN-TIME DATA REPRESENTATIONS

The Pascal-86 compiler determines the amount of storage needed at run-time for each data type, and the run-time support software allocates the storage when you execute the Pascal program. This appendix details the amount of storage allocated for each data type, and the method used to assign storage to these types and locate them in memory. You need this information if you intend to pass data as arguments to procedures written in other languages. See Appendix J for more information about communicating to modules written in other languages.

H.1 Simple Types

Table H-1 summarizes the storage allocated at run-time for each simple type. The figure under the BITS heading is the minimum number of bits needed to store values of the corresponding data type. The figure under the BYTES heading is the number of bytes allocated to store values of the corresponding data type. You only need to know the number of bytes for each data type in order to match a Pascal-86 data type with the appropriate data type in another Intel language.

In all of these discussions, a *byte* is eight bits, and a *word* is two consecutive bytes (16 bits). The byte with the higher memory address contains the eight most significant bits of a word, and the byte with the lower memory address contains the eight least significant bits. The significant bits of a value are right-justified in the bytes the value occupies. For all simple types, the run-time support software assigns the most significant bits of their binary representations to the highest bit positions. For real values, the run-time support software assigns the fractional part to the lowest bit positions (to the bytes with the lowest addresses), and the exponent part to the highest bit positions.

NOTE

All real arithmetic in Pascal-86 is performed in 80-bit TEMPREAL (extended precision) format, which is used both internally and as a predefined data type. Calculated values are converted to REAL or LONGREAL precision only upon assignment to such a variable. REAL values are in the intervals $(-2^{128}, -2^{-127})$, (0,0), and $(2^{-127}, 2^{128})$. LONGREAL values are in the intervals $(-2^{1024}, -2^{-1023})$, (0,0), and $(2^{-1023}, 2^{1024})$. TEMPREAL values are in the intervals $(-2^{16384}, -2^{-16383})$, (0,0), and $(2^{-16383}, 2^{16384})$. For further information on the interval formats for real values, see 7.1.8.

Decimal approximations for the powers of two are:

2^{-127} is approximately $5.877471754 \times 10^{-39}$
2^{128} is approximately $3.412823668 \times 10^{38}$
2^{-1023} is approximately $1.11253693 \times 10^{-308}$
2^{1024} is approximately $1.797693131 \times 10^{308}$
2^{-16383} is approximately $1.168105158 \times 10^{-4932}$
2^{16384} is approximately $1.189731472 \times 10^{4932}$

Table H-1. Run-Time Storage Allocation of Simple Data Types

Data Type	Bits	Bytes
BOOLEAN	1	1
CHAR[1]	8	1
INTEGER[2]	16	2
LONGINT[3]	32	4
LONGREAL[4]	64	8
REAL[5]	32	4
TEMPREAL[6]	80	10
WORD[7]	16	2
Pointer[8]	32	4
Subrange (*n..m*)[9]	*subbit*	*subbyte*
Enumerated (*n* elements)[10]	*enumbit*	*enumbyte*

NOTES:

[1] Values of type CHAR are eight-bit ASCII codes stored in eight bits (one byte). In Pascal programs, you can express the characters carriage return (ASCII 13) and line feed (ASCII 10) as the predefined constants CR and LF, respectively. All other characters can be expressed by enclosing the character in single quotes (ASCII 39). CHR is defined in the range 0 to 255.

[2] Values of type INTEGER are in the range −32767 to +32767.

[3] Values of type LONGINT are in the range −2147483647 to +2147483647.

[4] Values of type LONGREAL are in the intervals $(-2^{1024}, -2^{-1023})$, (0, 0), and $(2^{-1024}, 2^{1023})$ to 53 bits of precision.

[5] Values of type REAL are in the intervals $(-2^{128}, -2^{-127})$, (0, 0), and $(2^{-127}, 2^{128})$ to 24 bits of precision.

[6] Values of type TEMPREAL are in the intervals $(-2^{16384}, -2^{-16383})$, (0, 0), and $(2^{-16383}, 2^{16384})$ to 64 bits of precision.

[7] Values of type WORD are in the range 0 to 65535.

[8] If the SMALL(—CONST IN DATA—) model of segmentation is used (10.3.18) pointers are only 16 bits long (2 bytes).

[9] The value of *subbit* and *subbyte* can be calculated as follows:

If the subrange *n..m* is in the range 0..255, *subbit* is equal to eight bits and *subbyte* is equal to one byte; otherwise, *subbit* is equal to 16 bits, and subbyte is equal to two bytes.

[10] The value of *enumbit* and *enumbyte* can be calculated as follows:

For an enumeration of *n* elements, *enumbit* is equal to the smallest positive integer greater than or equal to the base-two logarithm of *n*, and *enumbyte* is equal to the smallest positive integer greater than or equal to (*enumbit*/8).

H.2 Structured Types

All structures are byte-aligned and occupy an integral number of bytes; i.e., each field of a structure starts at a bit position that is evenly divisible by eight and occupies a multiple of eight bits (bit positions are numbered from zero).

H.2.1 Record Types

The run-time support software assigns storage to record types in two steps: first the fields are assigned to a string of bits, and then the bit string is assigned to bytes in memory.

The fields of a record are assigned storage in the order in which they are defined in the record. The first field occupies the lowest address. Each field is byte-aligned and occupies an integral number of bytes.

Structured fields are aligned on the next available bit position that is evenly divisible by eight (byte-aligned), and they occupy an integral number of bytes (multiples of eight bits).

A structured type field can itself contain simple type and structured type fields, just as a record can contain records. If such a structure contains structured type or simple type fields, the structure is aligned to satisfy the byte-aligned requirement of its components.

The following figures illustrate the storage allocation scheme, which has two parts: first the fields are assigned to a long bit string, then the bit string is assigned to bytes of memory (shown in figure H-2).

```
TYPE A = RECORD
     A1    : 0..7;
     AREC  : RECORD  (*fields are byte-aligned*)
         A2  : INTEGER
         A3  : INTEGER
         END;
     A4    : 0..7
     END;
```

Figure H-1 shows the first step of this storage allocation scheme: assigning the fields to bytes in a long bit string.

```
 00000000 00111111 11112222 22222233 33333333 44444444
 01234567 89012345 67890123 45678901 23456789 01234567
+ ------ + ------ + ------ + ------ + ------ + ------ +
|   A1   |       A2        |        A3       |   A4   |
+ ------ + ------ + ------ + ------ + ------ + ------ +
```

Figure H-1. Record (A) Containing a Record (AREC) 121539-36

Storage for type A is allocated in this fashion ("+" denotes a byte boundary).

The second and last step is the assignment of the bit string to actual bytes, as shown in figure H-2. Within each byte, the more significant bits (higher bit positions) are actually on the *left* side, contrary to the bit numbering in figure H-1. Figure H-2 shows the actual bit positions of the fields defined in figure H-1 as they are assigned to bytes in memory.

```
 00000000 11111100 22221111 33222222 33333333 44444444
 76543210 54321098 32109876 10987654 98765432 76543210
+ ------ + ------ + ------ + ------ + ------ + ------ +
|   A1   |       A2        |        A3       |   A4   |
+ ------ + ------ + ------ + ------ + ------ + ------ +
```

Figure H-2. Actual Bit Positions within Memory Bytes 121539-37

H.2.2 Array Types

Since an array can contain arrays, the components of the innermost nested array are allocated first; that is, storage is allocated in row-major order, with the rightmost index expression varying the fastest. For example, a two-dimensional array of rows and columns can be simulated in Pascal-86 by providing an array of rows, where each row is an array of columns. The array of columns (selected by the rightmost index expression in a pair of index expressions) is allocated first.

The first component is assigned the lowest address. Each component is byte-aligned and occupies an integral number of bytes.

Using ASM86 to Reference an Element in a Pascal-86 Large Array

As mentioned in section 10.3.12, the data sections of certain modules may be placed in more than one data segment. In particular, arrays that require more than 64K bytes of data storage are placed in more than one data segment. These data structures are called large arrays.

This section describes the process whereby a large array is mapped onto memory segments. In addition, this section provides two examples that illustrate the mapping of large arrays, along with assembly source code that allows you to access a single element in each of these arrays.

Large arrays are mapped onto memory segments based on two static attributes of the structure: the component size and the component number. The component size of a large array is the number of bytes per individual component on the array. The component number is the total number of components on the array. The Pascal-86 compiler uses these numbers to map the elements of a large array onto a given number of segments. The first *n* segments are full; i.e., the number of elements they contain is a power of two. The final segment contains the remainder.

The algorithms presented in this section perform more efficiently when the component size of a large array is a power of two. This stems from a balance between the time needed to access an element in a large array and efficient memory usage. You may want to pad your data structure with unused bytes to construct a large array with a component size that is a power of two, thereby improving access time.

This section presents two examples in which a component in a large array is accessed. The first example presents a static array whose component size is a power of two; the second example presents a dynamically allocated array whose component size is not a power of two.

To partition the elements of a large array, the Pascal-86 compiler must first determine the total number of elements to place in a full segment. (A full segment is one whose component number is a power of two.) The following equation determines the number of elements in a full segment:

$$\text{LgNumc} = \text{Floor (Log base 2 (65536/Components Size))}$$

(65536 represents the maximum number of bytes in a given segment.)

Based on the results of this calculation, the compiler can partition elements into memory segments by calculating the number of bytes in a full segment and the number of bytes in the final segment of a large array. Another calculation that can be performed results in the number of full segments in a large array. Note that the Pascal-86 compiler performs these calculations; the equations are included here only to make the examples easier to understand.

These calculations are as follows:

1. Equation to calculate the number of bytes in a full segment:

   $$\text{Size of a full segment} = 2^{\text{LgNumc}} * \text{Component Size}$$

2. Equation to calculate the number of bytes in the last segment of a large array:

$$\frac{\text{(REM Component Number)}}{2^{\text{LgNumc}}} * \text{Component Size}$$

3. Equation to calculate the number of full segments in a large array:

$$\text{Number of Full Segments} = \text{MOD } \frac{\text{Component Number}}{2^{\text{LgNumc}}}$$

The following paragraphs present an example of the memory representation of a large array whose component size is a power of two.

Consider the large array representation for the following Pascal-86 program fragment:

```
Public ThisModule;
  For SomeOtherModule
    Var LargeArray: array[1..100000] of integer;
```

The number of elements in this array is 100,000 and the type is integer. Each integer in this 100,000-element array requires 2 bytes of storage; therefore, the total number of bytes needed to represent this array is 200,000.

Based on the equations presented above, this large array is represented in memory as three segments of 65536 bytes each and a final segment of 3392 bytes, as indicated in figure H-3.

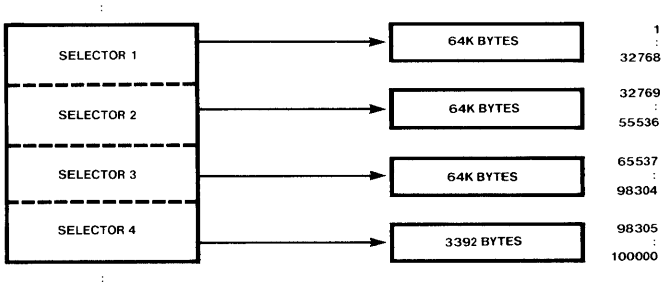


Figure H-3. The Memory Representation of a Large Array Whose Component Size Is a Power of Two

121539-45

The following section of source code (written in ASM86) allows you to access a single element in this static large array:

```
          ; assume normalized index expr in DX:AX

SAL   AX,1   ; Mult by Component Size (2 bytes)
RCL   DX,1
       ;  DX contains offset into selector table
       ;  AX contains offset into segment

MOV   DI,DX  ; Copy table index into index reg
SHL   DI,1   ; Each selector is 2 bytes, so (* 2)
MOV   ES,SelectorTableDisp [ DI ]

; ES contains correct selector for this element
```

```
MOV    DI,AX   ; Copy seg offset into index reg
MOV    CX,ES: [ DI ]

          ; CX contains the integer array element
```

The following paragraphs present an example of the memory presentation of a large array whose component size is not a power of two.

Consider the Pascal-86 representation for the following dynamically allocated large array whose component size is not a power of two:

```
Type
  ThreeByte =
    record
      field1 : 0..10;
      field2 : integer
    end;
  DynamicLargeArray = array [1..40000] of ThreeByte;
  Pntr = ↑DynamicLargeArray;
Var
  DynLAPtr : Pntr;
      :
begin
      :
  New(DynLAPtr);
      :
  Dispose( DynLAPtr );
      :
```

Here, the component size is three and the component number is 40,000. Based on the equations presented above, this large array is allocated to memory segments as two segments of 49152 bytes each and one remaining segment of 21696 bytes, as indicated in figure H-4.

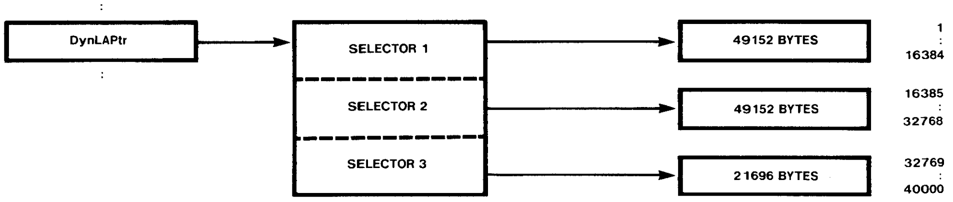


Figure H-4. The Memory Representation of a Large Array Whose Component Size Is Not a Power of Two 121539-46

The following section of source code (written in ASM86) makes it possible to access a single element in this large data array:

```
                  ; assume normalized index expr DX:AX
                  ; assume EA is in ES:BX
MOV    CX,14      ; LgNumC, used as shift count

MOV    SI,AX      ; save low 16 bits of index expr

                  ; compute selector table offset
                  ; first divide the index expr
                  ;                by 2**LgNumC
```

```
DIVIDELOOP :  SHR    DX,1   ; 32 bit division
              RCR    AX,1
     LOOP   DIVIDELOOP


     MOV    DI,AX    ; copy MOD into index reg
     SHL    DI,1     ; each selector is 2 bytes, so (* 2)


     MOV    ES,ES:[ BX + DI ]

    ; ES contains correct selector for this element

                     ; compute the offset into the seg

     AND    SI,0011111111111111B ; const is 2**LgNumC
     MOV    AX,3     ; constant is Component Size
     MUL    SI       ; component size * offset portion
     MOV    SI,AX    ; copy offset into index reg

   ; ES: [SI] is ADDRESS of selected array element

     MOV    CL,ES: [ SI ]  ; CL is arr[DX:AX].field1
```

H.2.3 Set Types

Sets may have at most 32767 elements, and the ordinal value of each member must lie in the range of integers.

The run-time support software stores a set such that the set element whose ordinal value is zero, modulo eight, is assigned to the least significant bit of the first byte (bit 0). The smallest set element in the base type of the set is also assigned to the first byte. Figure H-5 shows a sample assignment.

```
VAR ITEMS:   SET OF 3..10;
 .
 .
 .
ITEMS := [3,5,7,9];
 .
 .
 .
```

Storage for ITEMS is allocated in this fashion.

```
 76543210  76543210
+--------+--------+
|10101***|*****010|
+--------+--------+
```

Figure H-5. Bits Assigned for a Set 121539-38

The asterisk (*) denotes an unassigned bit and the plus sign (+) denotes a byte boundary.

H.2.4 File Types

A file variable occupies six bytes in its data area in addition to the space occupied by the buffer variable (which is allocated according to the rules for its type). Two of the six bytes hold the file's connection number, and the other four are used as a pointer to a structure in the constant area of the file variable's defining procedure.

The structure in the defining procedure's constant area consists of the name of the file variable, preceded by a byte holding the length of the name. The most significant bit of the length byte is set if the file is a program parameter.

Each component of a file starts on a byte boundary and occupies an integral number of bytes.

APPENDIX I PASCAL-86 EXTENDED SEGMENTATION MODELS

I.1 Introduction

Program *segmentation* is a technique used to optimize the object code produced by the compiler and to allow easier, more efficient memory addressing.

The simplest way to compile Pascal-86 code is to use the segmentation controls (LARGE, COMPACT, and SMALL) outlined in Chapter 10. However, to take full advantage of the segmented iAPX 86 architecture and to simplify the development of very large programs, Pascal-86 provides extended segmentation facilities. This appendix discusses the segmentation control options available to you.

Each compilation produces an object module made up of several sections (see 11.2). At link time, the sections from separately compiled modules are combined into segments depending on compiler-generated attributes and your input to the link program. Once combined into a segment, all constituent sections can be addressed from the same 8086 segment register. The compiler uses this segment addressability to improve the code produced for data references and procedure calls. This is the primary purpose of the segmentation controls—to tell the compiler how separately compiled code and data will be located in memory.

Most Pascal-86 programmers need not be concerned about memory addressing techniques on the iAPX 86, as the SMALL, COMPACT, and LARGE controls automatically handle the mechanics of program segmentation. If needed, a description of iAPX 86 memory concepts is given in Appendix J.

By making the variety of segmentation schemes open to you virtually unlimited, these controls can reduce the storage required for pointers and the code required to reference external variables and procedures.

I.1.1 Extended Segmentation

The segmentation controls adopted by Pascal-86 are a simple extension of the SMALL, COMPACT, and LARGE controls originally supported by PL/M-86. Using these simple controls, you can create a large program where each module is compiled with the same segmentation control, or you can use the extended controls to partition 'your program into a number of loosely-coupled subsystems.

A *subsystem* is a collection of tightly coupled, logically related modules that obey the same model of segmentation. A program is made up of one or more subsystems. (If you use only the simple controls to compile your program modules, then your program consists of one subsystem.) The subsystems within a program can use different segmentation models if appropriate. Within a subsystem, calls and data references are long or short depending on the segmentation model selected for the subsystem. Between subsystems, all calls are long and most data references require 32-bit pointers.

Optimized code is obtained by breaking LARGE programs (greater than 64K of code, data, or both) into COMPACT or SMALL subsystems having less than 64K each of code and data. References between modules in the same subsystem can then use more efficient 16-bit addresses. (Only references outside a subsystem need to use the full 32-bit address.) Use of these extensions also allows easier access to the one-megabyte address space.

I.2 Subsystems

By carefully constructing subsystems, you can minimize references to outside resources (code and data) and, in return, receive highly optimized code for all internal resource references. This section describes how to create subsystems using the extended segmentation controls.

Note that all modules in your program should be compiled with the same set of subsystem definitions, so that the compiler makes consistent assumptions about the location of externals. This can be done by putting the subsystem definitions and the interface specification into one INCLUDE file and inserting it in every compilation. (See figure I-1 for a sample INCLUDE file.) If inconsistent subsystem definitions are used when compiling modules, LINK86 will generate an error.

There is one segmentation control for each subsystem in a Pascal program. It takes the form:

```
$model ( [subsystem-id][submodel] HAS module-list [ { ; HAS module-list     } ] ... )
                                                  [ { ; EXPORTS public-list } ]
```

where

model specifies the model of segmentation that the subsystem will follow (SMALL, COMPACT, and LARGE are allowed, but LARGE subsystems will not provide optimized code). All modules in the subsystem must be compiled with the same model of segmentation.

subsystem-id specifies a unique name for each subsystem, and can be up to 31 characters long.

```
$SMALL(SmallHeap -CONST IN DATA- HAS WithSmallHeap;
$                EXPORTS TinyPtr;
$                EXPORTS WithSmallHeapNewLargeRec, NewLargeNumber;
$                EXPORTS WithSmallHeapGenerateTwo)
PUBLIC WithSmallHeap;
  FOR WithLargeHeap, Evaluator;
    TYPE TinyPtr = ^INTEGER;
  FOR WithLargeHeap;
    FUNCTION WithSmallHeapNewLargeRec (val1, val2 : integer) : LargeRecPtr;
    FUNCTION NewLargeNumber (val : integer) : LargePtr;
  FOR Evaluator;
    PROCEDURE WithSmallHeapGenerateTwo (FUNCTION c (rec1, rec2 : LargeRecPtr) : BOOLEAN;
                                        val1, val2 : INTEGER);

$LARGE(LargeHeap -CONST IN CODE- HAS WithLargeHeap, Evaluator;
$                EXPORTS LargePtr;
$                EXPORTS LargeRecPtr, LargeRec;
$                EXPORTS WithLargeHeapNewLargeRec, NewTinyNumber;
$                EXPORTS WithLargeHeapGenerateTwo;
$                EXPORTS compar)
PUBLIC WithLargeHeap;
  FOR WithSmallHeap, Evaluator;
    TYPE LargePtr = ^INTEGER;
         LargeRecPtr = ^LargeRec;
         LargeRec = RECORD
                      TinyVal : TinyPtr;
                      LargeVal : LargePtr
                    END;
  FOR WithSmallHeap;
    FUNCTION WithLargeHeapNewLargeRec (VAR val1, val2 : integer) : LargeRecPtr;
    FUNCTION NewTinyNumber (val : integer) : TinyPtr;
  FOR Evaluator;
    PROCEDURE WithLargeHeapGenerateTwo (FUNCTION c (rec1, rec2 : LargeRecPtr) : BOOLEAN;
                                        val1, val2 : INTEGER);

PUBLIC Evaluator;
  FOR WithSmallHeap, WithLargeHeap;
    FUNCTION compar (rec1, rec2 : LargeRecPtr) : BOOLEAN;
```

Figure I-1. INCLUDE File Containing Subsystem Definitions and Interface Specification

submodel specifies the placement of constants. It can be either:

-CONST IN CODE-

(for burning into ROM—default case for LARGE)

-CONST IN DATA-

(for efficient access to constants in programs that are loaded into RAM—default case for SMALL and COMPACT)

HAS *module-list* specifies all the modules that make up the subsystem. Each identifier in the module-list is the name of a module, which can be up to 31 characters long.

EXPORTS *public-list* lists the labels, procedures, and variables exported by this subsystem. Any object not named in an EXPORTS list will be local to its subsystem and not accessible from outside the subsystem. Each identifier in this list is the name of a public object, which can be up to 40 characters long.

Within a program, the subsystem name must be distinct from all module names, since they share the same name space. The names used to declare public objects must also be unique. Identifiers used to define subsystems and modules, however, may also be used to declare public objects.

In most applications of the subsystem controls, the HAS and EXPORTS lists will have several dozen entries apiece. To accommodate lists of this length, a subsystem control may be continued over more than one control line. (The continuation lines must be contiguous, and each must begin with a $ in the first column). Also, note that any number of HAS and EXPORTS lists may appear in a control, in any order, allowing you to format your subsystem specification so it can be easily read and maintained.

Consider the following subsystem definition:

```
$COMPACT(Room -CONST IN CODE- HAS Chair, Door, Window);
$COMPACT(Diner HAS Booths, Waitress, Jukebox ;
$              EXPORTS Lunches);
$LARGE(Allocate EXPORTS CodeSize, DataSize,
$                       MemSize, FreeSpace);
```

This sample program contains three subsystems: **Room**, **Diner**, and **Allocate**. **Room** and **Diner** use the COMPACT model of segmentation, while **Allocate** uses the LARGE model. Constants are stored with the code in **Room** and **Allocate**. The **Room** subsystem, containing the modules **Chair**, **Door**, and **Window**, is apparently the main program, since it exports no objects. **Allocate** supplies four objects: **CodeSize**, **DataSize**, **MemSize**, and **FreeSpace**.

I.2.1 Open and Closed Subsystems

The subsystems that make up your Pascal-86 program may be either open or closed. The subsystem definition for an *open* subsystem does not list the modules that it contains; it does not have a name. Modules can be added to this subsystem at any time without changing the subsystem definition. Each program may have only one open subsystem.

The subsystem definition for a *closed* subsystem does contain a HAS list, which specifies all modules in the subsystem. It also contains a *subsystem-id* as a name.

In an open subsystem, only the *submodel* and EXPORTS modifiers are permitted. By omitting the subsystem name, you automatically create an open subsystem that contains all modules not claimed in another subsystem's HAS list. (Note that use of the simple segmentation controls also creates an open subsystem.)

One advantage of using an open subsystem is that it simplifies dealing with a SMALL program whose code has grown too large. When your program exhausts its 64K code size, take a subset of your modules and put them into a closed subsystem, leaving the rest of the modules in the open subsystem.

For example, assume that your original program contained the modules **ATTACH**, **OPEN**, **CLOSE**, **ERRORS**, **ALLOCATE**, and **FREE**, which were all compiled with the simple control:

```
$SMALL
```

If you factor out the modules **ALLOCATE** and **FREE** from the original program, creating **SUBSYS1**, its subsystem definition would be:

```
$SMALL (SUBSYS1 HAS ALLOCATE, FREE)
```

Now, suppose that the modules remaining in the closed subsystem reference entry points **AllocBuff** and **FreeBuff** in **SUBSYBS1**. These must be exported from SUBSYS1 as follows:

```
$SMALL (SUBSYS1 HAS ALLOCATE, FREE;
$               EXPORTS AllocBuff, FreeBuff)
```

or:

```
$SMALL (SUBSYS1 HAS ALLOCATE; EXPORTS AllocBuff;
$               HAS FREE; EXPORTS FreeBuff)
```

The second form illustrates how multiple HAS and EXPORTS lists can be used to document the items exported from each module. It also illustrates the use of a continuation line.

Likewise, if a routine in **SUBSYS1** references the procedure **FatalError** in the module **ERRORS**, the definition of the open subsystem would then be:

```
$SMALL (EXPORTS FatalError)
```

No data structures need to be changed, since data reference values can still be 16 bits. All procedures will still use the short call and return mechanism, except for **AllocBuff**, **FreeBuff**, and **FatalError**.

I.2.2 The Exports List

A symbol included in a subsystem's EXPORTS list should be defined in the PUBLIC section of one of the modules in that subsystem. It is called an exported symbol, and may be referenced by modules in other subsystems. A public symbol defined within a subsystem, but not listed in its EXPORTS list, is called a domestic symbol. It may be referenced only by modules within the same subsystem.

A procedure should be exported only if it must be referenced outside the defining subsystem, since accessing exported procedures will generally require more code and more time than is normally required for domestic procedures.

Exported procedures have the following characteristics:

- They use the long form of call and return.
- They save and restore the caller's DS register upon entry and exit.
- They reload DS with their associated data segment upon entry.
- They accept VAR parameters using long or short addresses according to the segmentation model they were compiled with.

Public Symbols and the FOR-clause. As explained in 4.2.2, the FOR-clause defines the scope of public symbols in the interface specification. For domestic symbols, this scope may not extend beyond the defining subsystem. Hence, the FOR-clause for domestic symbols may name as recipients only modules that belong to the subsystem containing the symbol definitions. (Symbols not restricted by a FOR-clause can be accessed by all modules; consequently, only exported symbols should appear in an unrestricted definition list.) The FOR-clause for exported symbols may name any module in the program as a recipient.

I.2.3 Placement of Controls

The segmentation controls have special restrictions associated with their placement. These rules are as follows:

- Only the definition of the open subsystem (with no EXPORTS list) can be placed on the invocation line; definitions of all other subsystems must occur inside the source program.
- The subsystem definitions must appear before any PUBLIC section that is included in the subsystem.
- The definition of the open subsystem (if present) must be the last segmentation control specified.

The subsystem definitions for your entire program can be included in the compilation of each module using the INCLUDE control. The compiler will extract the information it needs to correctly and efficiently compile each module's intra- and inter-subsystem references.

Figure I-1 gives a sample INCLUDE file containing the segmentation controls and interface specifications for two subsystems, **SmallHeap** and **LargeHeap**.

Progamming Restrictions

Variable parameters of a procedure or function defined in a SMALL(—CONST IN DATA—) subsystem have restrictions on the objects that can be passed as arguments. They must be either variables defined in a SMALL subsystem, dynamic variables whose pointer types were defined in SMALL(—CONST IN DATA—) subsystems, or variable parameters to a procedure defined in a SMALL (—CONST IN DATA—)subsystem.

Variable parameters of a procedure or function in a SMALL(—CONST IN DATA—)subsystem use short addresses. Corresponding arguments must be variables in a SMALL(—CONST IN DATA—) subsystem or referenced through a 16-bit pointer (type defined in SMALL (—CONST IN DATA—) subsystem).

Any other invalid mixing of the short and long pointers will be prohibited by the strong typing mechanism in Pascal.

APPENDIX J
LINKING TO MODULES WRITTEN IN OTHER LANGUAGES

This appendix describes the calling conventions used by iAPX 86,88 family languages. These calling conventions are standardized so that a main module written in Pascal-86 can freely call procedures, subroutines, and subprograms in other modules written in other iAPX 86,88 family languages. (For information on coding main program modules in other iAPX 86 languages, such as FORTRAN-86 or PL/M-86, see J.6.)

As a Pascal-86 programmer calling Pascal-86 procedures and functions from Pascal-86, you do not need the information in this appendix. You need to know information about parameters and arguments as described in Chapter 6.

As a Pascal-86 programmer calling FORTRAN-86 subprograms from Pascal-86, you must know the FORTRAN-86 data types that match Pascal-86 data types, and the order and number of arguments to supply for the FORTRAN-86 parameters. A table of the corresponding data types is provided at the end of this appendix (J.5). Note that FORTRAN uses call by reference for all parameters, which corresponds to VAR parameters in Pascal. (FORTRAN-86 programs conform to the LARGE model of segmentation.)

As a Pascal-86 programmer calling PL/M-86 procedures from Pascal-86, you must read the special note in J.4.1 (Stack Usage). You must also know the PL/M-86 data types that match Pascal-86 data types, and the order and number of arguments to supply for the PL/M-86 parameters. A table of the corresponding types is provided at the end of this appendix.

As a Pascal-86 programmer calling ASM86 subroutines or linking to the data in 8086 Macro Assembly Language programs from Pascal-86, you need to know the calling conventions of stack and register usage described in this appendix. You also need to know the corresponding data types listed at the end of this appendix in order to write a subroutine that can pick up the data your Pascal-86 program passes to it. Refer to the *ASM86 Macro Assembler Operating Instructions* for more information about linking to ASM86 programs and for examples of linking such programs to PL/M-86 programs. (Since Pascal-86 and PL/M-86 use the same calling conventions, these examples also apply when linking ASM86 programs with Pascal-86 programs.)

As a Pascal-86 programmer, PL/M-86 programmer, or macro assembly language programmer, you also need to know how to link modules properly, as described in Chapter 12 and in the *iAPX 86,88 Family Utilities User's Guide.*

J.1 Introduction

Since Pascal-86 allows you to write and compile separate Pascal-86 modules that can be linked together at a later time, you can solve a big problem with a solution composed of separately-tested modules that are linked together *after* they are internally bug-free. Not all modules have to be in Pascal-86—you can choose the appropriate language for each module. (Information on coding the main module in other languages is given in J.6.) Be sure to link the modules properly with LINK86, the 8086-based linker (to satisfy references to externals). Since the iAPX 86,88 family languages follow the same calling sequence (described in the following section), control will pass to a called module correctly. However, the called module might not be able to deal intelligently with the data passed to it, because languages treat some data structures differently.

NOTE

> The term *procedure* is used in Pascal and in PL/M, *subprogram* in FORTRAN, and *subroutine* in assembly language. In this appendix, *subprogram* denotes any entity written in any iAPX 86,88 language that can call a Pascal-86 procedure or function or be called from a Pascal-86 module.

The interface specification of a Pascal-86 module contains that module's public section and the public sections of other modules that communicate with the Pascal-86 module. Public sections are explained in Chapter 4. You have to define the public objects which the other modules refer to, and the external objects that can be referred to by the Pascal-86 module. These objects include the names of external procedures and functions that can be called from the module, and public procedures and functions in the module that can be called from other modules.

By specifying arguments in a reference to an external subprogram (procedure, function, or subroutine), you pass data to the external subprogram. The number of arguments and the order in which you specify them must match the number and order of the corresponding parameters in the external subprogram's declaration (see 6.1).

All arguments for parameters are passed on the 8086 stack or the 8087 register stack in the order that they are specified. Functions that return values return a simple-type value (except real values) in a register, or a real value on the top of the 8087 register stack. Pascal-86 functions do not return structured values.

There are two methods of passing arguments to other subprograms: call by value and call by reference. The first method, *call by value*, passes the actual value of the argument to the subprogram. The second method, *call by reference*, passes the *address* of the argument to the subprogram. The called subprogram uses the address to find the data structure associated with the argument. In both cases, the called subprogram must know the structure of the data.

Pascal-86 passes arguments to variable parameters by reference, and arguments to value parameters by value.

J.2 iAPX 86 Memory Concepts

The allocation and arrangement of run-time program memory (via relocation and linkage) depend on the size control (SMALL, COMPACT, or LARGE) that you specified when compiling your program modules. These controls also influence how locations are referenced in the compiled program, leading to certain program size restrictions.

Each compilation produces an object module containing several sections, for example, code, data, and stack (see 11.2). At link time, sections from separate modules can be combined into segments, depending on compiler-generated attributes and user input to the link program. Once they are combined into a segment, all the sections can be addressed without reloading an iAPX 86 segment register.

iAPX 86 memory space has an extent of one megabyte, but the 16-bit word length of the 8086 can only address 64K locations. A complete physical address requires 20 bits. Therefore, two separate words (a segment address and an offset) are used in a special way to form this 20-bit address, as follows:

- A *segment* is defined as a portion of the 8086's one megabyte address space, consisting of up to 64K bytes of contiguous memory and beginning at a 16-byte boundary. Thus, the hexadecimal representation of the 20-bit address for the beginning of every segment ends in 0, for example, 00000H, 00010H,...12340H,.... This definition permits a 16-bit word to represent any segment starting address since the extra four bits not included are always 0. A 16-bit word used in this way is called a *segment address*. Four CPU registers (CS, DS, SS, and ES) are used to hold segment addresses.

- The second word used to form the full 20-bit address identifies a specific location within the segment, starting at the segment address. This 16-bit quantity is called the *offset*.

To form a 20-bit address, the iAPX 86 CPU shifts a segment address left four bits and adds an offset.

J.3 Segment Name Conventions

Table J-1 summarizes the segmentation of a subsystem under the possible models of segmentation by giving the name of the segment and group where each type of program section is stored for each model of segmentation.

J.4 Calling Sequence

The calling sequence for each subprogram activation (procedure reference, CALL, or function reference) places the arguments for the subprogram parameters on the 8086 stack or 8087 register stack and then activates the subprogram with a CALL instruction. You can see an approximate assembly codc listing of this sequence by

Table J-1. Summary of Pascal-86 Segment and Group Names

Model	SubModel	Code	Data Static	Data Heap	Data Memory	Constants	Pointers
SMALL	IN DATA	*s*CODE *s*CGROUP	*s*DATA DGROUP	MEMORY DGROUP	MEMORY DGROUP	*s*CONST DGROUP	16 bits
SMALL	IN CODE	*s*CODE *s*CGROUP	*s*DATA DGROUP	dynamic	MEMORY DGROUP	*s*CODE *s*CGROUP	32 bits
COMPACT	IN DATA	*s*CODE *s*CGROUP	*s*DATA *s*DGROUP	dynamic	MEMORY ——	*s*CONST *s*DGROUP	32 bits
COMPACT	IN CODE	*s*CODE *s*CGROUP	*s*DATA *s*DGROUP	dynamic	MEMORY ——	*s*CODE *s*CGROUP	32 bits
LARGE	IN DATA	*m*CODE ——	*m*DATA ——	dynamic	MEMORY ——	*m*DATA ——	32 bits
LARGE	IN CODE	*m*CODE ——	*m*DATA ——	dynamic	MEMORY ——	*m*CODE ——	32 bits

NOTES:

*s*CGROUP denotes a group name composed of the subsystem name and _CGROUP
*s*DGROUP denotes a group name composed of the subsystem name and _DGROUP
*s*CODE denotes a segment name composed of the subsystem name and _CODE
*s*DATA denotes a segment name composed of the subsystem name and _DATA
*m*CODE denotes a segment name composed of the module name and _CODE
*m*DATA denotes a segment name composed of the module name and _DATA

compiling with the CODE control a Pascal-86 module that contains a reference to an external procedure or function.

J.4.1 Stack Usage

The arguments are placed on the 8086 stack or the 8087 register stack in left-to-right order. Since the stack grows from higher locations to lower locations, the first argument occupies the highest position on the stack.

In a list of arguments for value parameters, the leftmost seven real argument values (if any) are passed on the 8087 register stack, with each argument value occupying one 80-bit register. If there are more than seven real argument values, the rest (after the leftmost seven) are passed on the 8086 stack.

Three 8086 stack words are required to hold arguments for functional and procedural parameters. The first two words hold a four-byte address of the code, and the third word holds a two-byte static link to the environment of the code.

NOTE

In PL/M-86, arguments for procedural parameters have no static link to the environment of the code, only a four-byte address of the code. There are two restrictions on the use of procedural or functional arguments passed between Pascal-86 and PL/M-86 subprograms:

1. The PL/M-86 subprogram must declare a dummy WORD parameter immediately after the procedural or functional parameter, in order to accommodate the stack space occupied by the static link to the environment of the code.
2. You can only pass outermost procedures and functions (those not nested within other procedures or functions) between PL/M-86 and Pascal-86 subprograms.

When an assembly language program calls a Pascal-86 procedure or function, the assembly language program must first push the arguments onto the stack. For example, suppose an assembly language program is calling the Pascal-86 procedure PROC1, which is declared in the PUBLIC section of the Pascal-86 module as follows:

```
PROCEDURE proc1(PARM1,PARM2,PARM3:INTEGER;
                REALPARM1,REALPARM2:REAL);
```

The assembly language program must push the argument for **PARM1** first, the argument for **PARM2** second, and the argument for **PARM3** third, onto the 8086 stack, and push the arguments for **REALPARM1** and **REALPARM2** onto the 8087 register stack since they are REAL parameters. (The seven leftmost real arguments in an argument list are passed on the 8087 register stack.)

When a Pascal-86 program calls an assembly language subprogram, the program must use a reference to the name of the assembly language subprogram, and supply an argument list in a left-to-right order that corresponds to the first-to-last order of parameters expected by the assembly language subprogram.

Figure J-1 shows the state of the 8086 and 8087 stacks after a call is made to the Pascal-86 procedure **PROC1**. The stack layouts would be the same if a Pascal-86 procedure called an assembly language subprogram which expected the parameters **PARM1**, **PARM2**, **REALPARM1**, **PARM3**, and **REALPARM2** in that order.

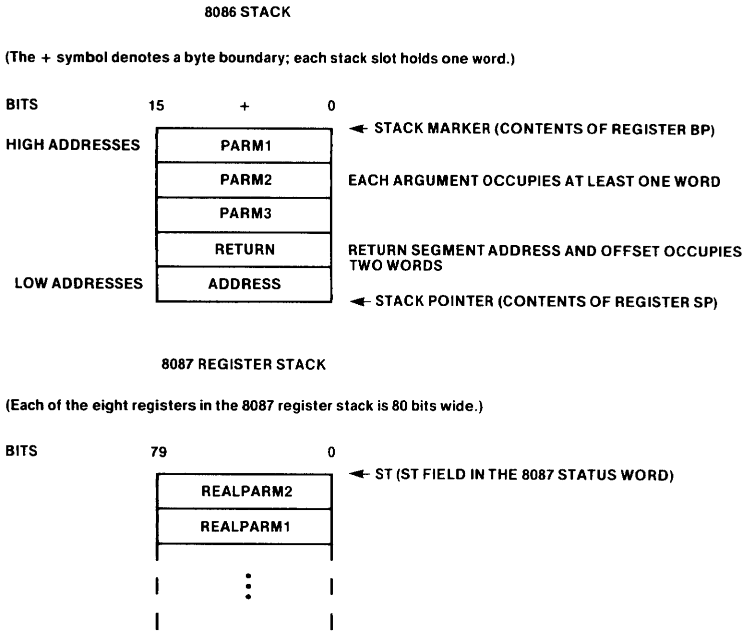

8086 STACK
(The + symbol denotes a byte boundary; each stack slot holds one word.)
BITS
15
+
0
HIGH ADDRESSES
PARM1
PARM2
PARM3
RETURN
ADDRESS
LOW ADDRESSES
◄ STACK MARKER (CONTENTS OF REGISTER BP)
EACH ARGUMENT OCCUPIES AT LEAST ONE WORD
RETURN SEGMENT ADDRESS AND OFFSET OCCUPIES TWO WORDS
◄ STACK POINTER (CONTENTS OF REGISTER SP)
8087 REGISTER STACK
(Each of the eight registers in the 8087 register stack is 80 bits wide.)
BITS
79
0
REALPARM2
REALPARM1
◄ ST (ST FIELD IN THE 8087 STATUS WORD)

Figure J-1. 8086 and 8087 Stack Layouts When Subprogram Is Activated

121539-39

Each 8086 stack slot holds one word. Single-byte arguments are pushed onto the 8086 stack as words. Single-byte arguments are not sign-extended when pushed onto the 8086 stack (the high-order byte is undefined). Arguments for functional and procedural parameters occupy three 8086 stack words: the first two for the four-byte address of the code, and the third for a two-byte static link to the environment of the code.

Each REAL argument occupies one 8087 stack register. There are only eight registers in the 8087 stack; therefore, if more than seven REAL arguments are passed in one call, the leftmost (first) seven are passed on the 8087 stack, and the rest are passed on the 8086 stack.

For other arguments to value parameters, space on the 8086 stack is allocated according to their type. For example, records, arrays, and sets will usually require more than two bytes; see Appendix H for run-time data representations. Arguments may take up to 65535 bytes; consequently, any data structure can be passed by value on the stack.

A function (or PL/M-86 typed procedure) that returns a simple value returns the value in a register as described in the next section. A real value is returned on the 8087 register stack; the real value occupies the top 8087 stack register.

A Pascal-86 program that calls an assembly language subprogram expects the stack to look the same after the subprogram returns as it looked before the subprogram was called; that is, before arguments were pushed onto the stack. A called assembly language subprogram can restore the stack to its former condition by using the RET *n* instruction, where *n* is the number of *bytes* occupied by the arguments passed to it.

An assembly language program can expect the stack, upon return from a Pascal-86 subprogram, to no longer contain the arguments it pushed, because the Pascal-86 subprogram's object code adjusts the Stack Pointer (contents of the SP register).

Figure J-2 shows the 8086 stack layout during the execution of a called subprogram.

J.4.2 Register Usage

When an external subprogram is called, the register contents are as shown in table J-2.

In addition, the 8087 register stack contains the first seven REAL arguments passed by the calling program. The 8087 status word is unknown and need not be saved, and the 8087 mode word must be saved on entry and restored before exit, if it is changed in the called subprogram.

If an assembly language subprogram expects to be called by a Pascal-86 program, and the subprogram alters the segment registers DS or SS, the subprogram must save the contents of these registers upon entry and restore them prior to returning to the Pascal-86 program.

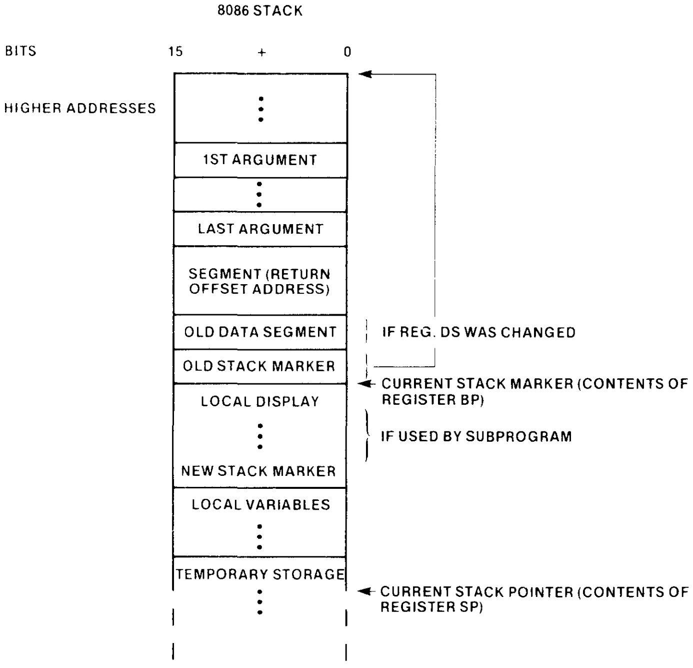


Figure J-2. 8086 Stack Layout during Subprogram Execution 121539-40

Table J-2. 8086 Register Contents When Calling an External Subprogram

Register	Contents
AX	Unknown; need not be saved.
BX	Unknown; need not be saved.
CX	Unknown; need not be saved.
DX	Unknown; need not be saved.
BP	Calling program's stack marker.
SP	Top of stack pointer.
SI	Unknown; need not be saved.
DI	Unknown; need not be saved.
DS	Calling program's data segment (static link).
CS	Called program's code segment.
SS	Calling program's stack segment.
ES	Unknown, need not be saved.

Pascal-86 uses the BP register to address the stack. A called assembly language subprogram must save the contents of the BP register upon entry, and restore its contents before returning control to the Pascal-86 program, if the called subprogram uses the BP register. Before returning, the called subprogram must also adjust the SP register to remove all parameters from the 8086 stack.

The registers AX, BX, CX, DX, SI, DI, and ES do not need to be preserved. A called assembly language subprogram can freely use these registers. If the called subprogram returns values, they are returned in the registers described in the next section.

An assembly language program calling a Pascal-86 subprogram cannot expect the contents of the 8086 general-purpose registers, except BP and SP, to be preserved. If they are needed, they must be saved prior to calling the Pascal-86 subprogram.

Table J-3 summarizes the use of registers.

Table J-3. Summary of 8086 Register Usage

Register	Must Preserve	Usage
AX	no	Return BYTE (AL), WORD, and INTEGER values.
BX	no	Return POINTER offset.
CX	no	—
DX	no	—
SP	yes*	Stack pointer.
BP	yes	Stack marker.
SI	no	—
DI	no	—
FLAGS	no	—
CS	no	Called subprogram's code segment.
DS	yes	Caller's data segment.
SS	yes	Caller's stack segment.
ES	no	Return POINTER segment address.

*SP must be adjusted so that all arguments are removed from the stack upon return.

J.4.3 Returned Values

Pascal-86 functions, PL/M-86 typed procedures, and other called subprograms that return a value to the calling program use the registers to hold simple data types. REAL values are returned on the 8087 register stack.

Table J-4 shows the registers used to return values.

Table J-4. Registers Used to Return Simple Values

Register	PL/M-86 Type	Fortran-86 Type	Pascal-86 Type
8086:			
AL	BYTE	INTEGER*1 LOGICAL*1	CHAR, BOOLEAN, unsigned subrange, or enumeration stored in 8 bits.
AX	INTEGER, WORD, or SELECTOR	INTEGER*2 LOGICAL*2	INTEGER, WORD, subrange, or enumeration stored in 16 bits.
DX:AX	DWORD	INTEGER*4 LOGICAL*4	LONGINT

ES(segment)
BX(offset)
	POINTER (all models except SMALL RAM)		Pointer (all models except SMALL(—CONST IN DATA—))

BX(offset only)	POINTER (SMALL RAM)		Pointer (SMALL (—CONST IN DATA—) model)
8087:			
ST	REAL	REAL	REAL, LONGREAL, TEMPREAL

J.4.4 NEAR and FAR Procedures

To call a Pascal-86 subprogram from assembly language, your ASM86 procedure must declare the Pascal procedure to be either NEAR or FAR. The choice depends on the segmentation model used to compile the module containing the Pascal-86 subprogram, and whether or not the subprogram is exported from its subsystem.

To call an assembly language subprogram from Pascal-86, you must provide a Pascal definition of the external procedure in one of the PUBLIC sections in the interface specification.

Use the following chart to determine the appropriate attribute for the ASM86 subprogram.

Model	Exported	Domestic
SMALL	FAR	NEAR
COMPACT	FAR	NEAR
LARGE	FAR	FAR

J.4.5 Example: Pascal-86 Calling an Assembly Language Subprogram

The following simple example shows a partial Pascal-86 program calling an ASM86 subprogram SUBPRG, as shown in figure J-3. This example assumes that the default segmentation model LARGE is used. For examples using the other models, see the *ASM86 Macro Assembler Operating Instructions.*

```
system-id  8086/87/88/186 MACRO ASSEMBLER V2.0 ASSEMBLY OF MODULE EXMPL
OBJECT MODULE PLACED IN EXMPL.OBJ
ASSEMBLER INVOKED BY:  ASM86.86 EXMPL.ASM


LOC  OBJ                     LINE     SOURCE

                               1      NAME EXMPL
                               2      ;
                               3      ; This program demonstrates procedure linkage to PASCAL-86,
                               4      ; focusing on the parameter passing conventions.
                               5      ;
                               6      ; This procedure takes five arguments for five parameters:
                               7      ; a byte, a word, an integer, an integer variable and
                               8      ; an integer function.
                               9      ; They are pushed onto the stack in that order.
                              10      ; They must be popped at exit (with the RET instruction).
                              11      ;
                              12      ; The prologue code saves BP, and points BP to the
                              13      ; structure defined below.  After the prologue executes,
                              14      ; the stack looks like this:
                              15      ;
                              16      ;         high memory
                              17      ;      ----------------
                              18      ;      |    PARM1     | ----> argument A
                              19      ;      ----------------
                              20      ;      |    PARM2     | ----> argument B
                              21      ;      ----------------
                              22      ;      |    PARM3     | ----> argument C
                              23      ;      ----------------
                              24      ;      |  (segment)   |}
                              25      ;      -----PARM4-------}----> argument D
                              26      ;      |  (offset)    |}
                              27      ;      ----------------
                              28      ;      |  (segment)   |}
                              29      ;      -----PARM5------}
                              30      ;      |  (offset )   |}----> argument FUNC
                              31      ;      -----PARM5------}
                              32      ;      |(environment) |}
                              33      ;      ----------------
                              34      ;      |  (segment)   |
                              35      ;      -Return Address-
                              36      ;      |  (offset )   |
                              37      ;      ----------------
                              38      ;      |   old DS     |
                              39      ;      ----------------   } saved in prologue
                              40      ;      |   old BP     |
                              41      ;      ----------------   <---SP, BP point to here
                              42      ;         low memory
                              43      ;
                              44      ; The required STRUCTURE definition is:
                              45      ;
----                          46      DSA     STRUC
                              47
0000                          48      OLD_BP  DW        ?    ; Prologue code saves BP here
0002                          49      OLD_DS  DW        ?    ; Prologue code saves DS here
0004                          50      RETURN  DD        ?    ; Double word for FAR procedures
0008                          51      ENVIRON DW        ?    ; PARM5 static link to environment
000A                          52      CODEPTR DD        ?    ; Pointer to code of PARM5 function
000E                          53      PARM4   DD        ?    ; Pointer to variable
0012                          54      PARM3   DW        ?    ; A PASCAL-86 integer value
0014                          55      PARM2   DW        ?    ; A PASCAL-86 enumeration in 16 bits
0016                          56      PARM1   DB        ?    ; but is stored on stack in one word,
0017                          57      XXX     DB        ?    ; with the high order byte undefined
                              58
----                          59      DSA     ENDS
                              60
                              61      ; Inside the subprogram, value arguments are accessed simply by
                              62      ; using a STRUCTURE reference, with BP as the base, and the
                              63      ; appropriate field name as the qualifier; example: [BP].PARM3.
                              64      ;
                              65      ; NOTE: The STRUCTURE fields for the arguments are declared in
                              66      ; reverse order in which they were pushed, due to the fact
                              67      ; the 8086 stack grows towards low memory.
                              68      ;
                              69      ; The saved value of BP and the return address must be declared
                              70      ; in the structure, since these two items are pushed between the
                              71      ; arguments and the spot pointed to by BP.
                              72      ;
```

Figure J-3. An ASM86 Subprogram Called from Pascal-86

```
----                SUBPRG_DATA    SEGMENT          ; not combinable
0000 ??               A_LOCAL      DB          ?    ; local variables go here
----                SUBPRG_DATA    ENDS

----                SUBPRG_CODE    SEGMENT          ; not combinable
                    ;
                    ; SUBPRG does nothing but call the function PARM5 and access
                    ; the first four arguments.  The prologue code saves BP, and
                    ; then copies SP to BP, allowing the value arguments to be
                    ; picked up conveniently with the BP register.
                    ;
                            PUBLIC SUBPRG
0000                SUBPRG PROC   FAR
0000 1E                     PUSH   DS                       ; prologue code, preserve DS
0001 B8----       R         MOV    AX,SUBPRG_DATA           ; address local data segment
                            ASSUME CS:SUBPRG_CODE, DS:SUBPRG_DATA
0004 55                     PUSH   BP                       ; preserve BP for PASCAL-86
0005 8BEC                   MOV    BP,SP

                    ; call the function argument PARM5
0007 55                     PUSH   BP                       ; save BP across call
0008 8BF5                   MOV    SI,BP
000A 8B6E08                 MOV    BP,[BP].ENVIRON          ; establish static link
000D 36FF5C0A               CALL   SS:[SI].CODEPTR          ; indirect call to parms
0011 5D                     POP    BP

0012 8A4E16                 MOV    CL,[BP].PARM1            ; PARM1 is at BP+22
0015 8B5614                 MOV    DX,[BP].PARM2            ; PARM2 is at BP+20
0018 8B4612                 MOV    AX,[BP].PARM3            ; PARM3 is at BP+18
001B C45E0E                 LES    BX,[BP].PARM4            ; ptr to PARM4 is at BP+14
001E 268B07                 MOV    AX,ES:[BX]               ; access PARM4

0021 5D                     POP    BP
0022 1F                     POP    DS
0023 CA1000                 RET    16                       ; return and POP 16 parameter bytes

                    SUBPRG         ENDP
----                SUBPRG_CODE    ENDS
                                   END

ASSEMBLY COMPLETE, NO ERRORS FOUND
```

Figure J-3. An ASM86 Subprogram Called from Pascal-86 (Cont'd.)

The interface specification for the Pascal-86 program must contain a reference to the external procedure SUBPRG:

```
PUBLIC SEPARATEMOD;
PROCEDURE SUBPRG(bval:CHAR;
                 wval:objects;
                 ival:INTEGER;
                 VAR ivar:INTEGER;
                 FUNCTION ff: INTEGER);
```

The value parameters for **SUBPRG** are **BVAL** (one byte), **WVAL** (one word) of type "objects" ("objects" is defined elsewhere as an enumeration of greater than 256 but less than 32768 elements), and **IVAL** of type INTEGER. In addition, SUBPRG has a variable parameter **IVAR** of type INTEGER, and a functional parameter **FF** of type INTEGER. Assume that the Pascal-86 program assigns values to variables, and uses those variables and a function reference as arguments (A, B, C, D, and FUNC) in the statement that calls **SUBPRG**:

```
SUBPRG(A,B,C,D,FUNC); (* Call SUBPRG with A,B,C,D,FUNC *)
```

The arguments **A**, **B**, and **C** are passed using *call by value* to satisfy the parameters **BVAL**, **WVAL**, and **IVAL**. The arguments **D** and **FUNC** are passed using *call by reference* to satisfy the parameters **IVAR** and **FF**. The assembly language subprogram **SUBPRG** picks up the values of **A**, **B**, and **C**, the reference to **D**, and the address and static link to **FUNC**, by using an ASM86 structure to describe the stack, as shown in figure J-3.

J.5 Compatible Data Types

Table J-5 presents the compatible PL/M-86, ASM86, and FORTRAN-86 data types for each Pascal-86 type. For run-time data representations of Pascal-86 types, see Appendix H.

Table J-5. Data Types Compatible with Pascal-86 Data Types

Pascal-86 Data Type	PL/M-86	ASM86	Fortran-86
CHAR, enumeration, unsigned subrange, or set stored in 8 bits	BYTE	DB	none
BOOLEAN	BYTE	DB	LOGICAL*1
INTEGER or subrange stored in 16 bits	INTEGER	DW	INTEGER*2
WORD, enumeration, or set stored in 16 bits, or subrange 0..64K-1	WORD	DW	none
LONGINT	none	none	INTEGER*4
Pointer (all models except SMALL(—CONST IN DATA—))	POINTER (all models except SMALL(—CONST IN DATA—))	DD	none
Pointer (SMALL(—CONST IN DATA—) model)	POINTER(SMALL (—CONST IN DATA—) model) or WORD	DW	none
REAL	REAL	DD	REAL
LONGREAL	none	DQ	REAL*8 or DOUBLE PRECISION
TEMPREAL	none	DT	TEMPREAL
ARRAY(*m..n*) of same base type	ARRAY	none	ARRAY**
RECORD of fields of matching type	STRUCTURE	STRUC	none
PROCEDURE (argument definitions must match)	*PROCEDURE (argument definitions must match)	PROC (no argument definitions)	SUBROUTINE (argument definitions must match)
FUNCTION (argument definitions and return types must match)	*PROCEDURE (argument definitions and return types must match)	none	FUNCTION (argument definitions and return types must match)

* See note, J.4.1 (Stack Usage).

** Note that multi-dimensional arrays in Fortran-86 have their dimensions reversed relative to Pascal-86 arrays.

J.6 Coding the Main Module in Other Languages

Pascal-86 performs I/O and floating-point initialization right in the main module. Consequently, when combining PASCAL modules with modules written in other languages, it is expected that the main module is written in Pascal. However, you may want to code the main module in another language. If you code it in PL/M-86 or ASM86, you must follow all five steps given below. If you want to code the main module in FORTRAN-86, you need only perform steps 2, 3, and 4.

1. Initialize the Universal Transput System (UTS). To do this, the main module must call two parameterless external procedures, INITFP (or INIT87) and TQ_001. These perform floating-point and I/O initialization, respectively. (Note that INITFP unmasks the invalid exception, while INIT87 masks it.)

2. Allocate file variables for the predefined text files INPUT and OUTPUT, if necessary. All other files to be used in the Pascal portions of your system should be declared in the PUBLIC section of one of the Pascal modules. To allocate file variables for the INPUT and OUTPUT files, use the names PQ_INPUT and PQ_OUTPUT, respectively. Include declarations of these file variables in the PUBLIC section of one of your Pascal modules, as follows:

   ```
   PUBLIC PASCMOD;
     VAR PQ_INPUT, PQ_OUTPUT: TEXT;
   ```

3. Initialize global Pascal files. To do this, the main module must set the first six bytes of each file variable to zero.
4. Call a Pascal program that resets (opens) PQ_INPUT and PQ_OUTPUT to specific files and devices. To open these files, use the Pascal-86 built-in routines RESET and REWRITE. (RESET and REWRITE must be called from a routine coded in Pascal, since they do not follow the standard calling sequence).

   ```
   RESET (PQ_INPUT, ':CI:');
   REWRITE (PQ_OUTPUT, ':CO:');
   ```

5. After the Pascal program has ended, call TQ_999, another parameterless external procedure, which performs I/O close-down.

Figure J-4 is an example of a main module coded in PL/M-86; Figure J-5 is a sample Pascal-86 subroutine called from the main module. Note that LINK86 generates three "type mismatch" warnings that may be ignored.

```
$LARGE
main:do;

 declare realfile(6) byte external;
 declare (pq_input, pq_output)(6) byte external;
 declare i integer;
 declare y real;
 writeit: procedure(x) external;
 declare x real;
 end writeit;

 initfp: procedure external;
 end initfp;

 tq_001: procedure external;
 end tq_001;

 tq_999: procedure external;
 end tq_999;

 do;
  do i = 0 to 5;
   realfile(i) = 0;
   pq_input(i) = 0;
   pq_output(i) = 0;
  end;
  call initfp;
  call tq_001;
  y = 10.5;
  call writeit(y);
  call tq_999;
 end;
end main;
```

Figure J-4. PL/M-86 Main Module Calling Pascal-86 Subprogram

```
module ProcWithIO;

public ProcWithIO;
   var PQ_INPUT, PQ_OUTPUT: text;
       RealFile: text;
   procedure Writeit(x:real);
private ProcWithIO;

procedure Writeit(x:real);
begin
  reset(PQ_INPUT, ':CI:');
  rewrite(PQ_OUTPUT, ':CO:');
  rewrite(RealFile, 'echo');
  writeln('PARAMETER PASSED IS ', x:8:4);
  write('INPUT ANOTHER REAL NUMBER');
  readln(x);
  writeln('THE NUMBER YOU ENTERED IS', x:8:4);
  writeln(RealFile, x:8:4)
end;
.
```

Figure J-5. Pascal-86 Subprogram Called from PL/M-86 Main Module

APPENDIX K
RUN-TIME INTERFACE

This appendix describes the run-time system that supports Pascal-86 programs, and shows how to interface an operating environment to the run-time system in order to execute Pascal-86 programs. Depending on the operating system you are using, you may be able to disregard this information. See your specific host-system appendix for details.

The first section describes run-time interrupt handling and provides background information. You can use this information to supply your own interrupt handlers. (See your specific host-system appendix for information on real arithmetic interrupt handling.)

The second section describes Pascal-86 run-time storage management. This includes a discussion of the heap mechanism for SMALL programs (see 10.3.18). You can use this information to supply your own memory managers.

The third section describes the *logical record interface*, which is a set of procedure names and calling conventions that provide an interface between the Pascal-86 run-time support software and an operating system. You can provide your own procedures using these names and calling conventions in order to hook your own operating system or operating environment to this interface, and thereby use the Pascal-86 run-time software along with Pascal-86 programs in your own operating environment.

A more detailed discussion of the run-time support system can be found in the *iAPX 86,88 Run-Time Support Manual.*

K.1 Run-Time Interrupt Processing

You need to read this section only if you intend to provide your own interrupt handlers and/or override the default handlers.

There are two interrupt pins on the 8086 processor: the "non-maskable interrupt" pin (NMI) and the "maskable interrupt" pin (INTR). The "non-maskable interrupt" cannot be ignored by the processor, whereas the "maskable interrupt" can be enabled or disabled.

Each "maskable interrupt" has an *interrupt number* that designates the type of interrupt. For example, interrupt number 4 tells the processor that an interrupt of type 4 (integer overflow) is occurring, and the processor looks for the procedure associated with interrupt number 4 in order to execute the procedure to handle the interrupt.

Interrupt numbers range from 0 to 255. Interrupt number 0 is reserved for integer divide-by-zero errors. Interrupt numbers 1 through 3 are reserved for single stepping, "non-maskable interrupts," and the INT instruction, respectively. Interrupt number 4 is reserved for integer overflow, and integer number 5 is reserved for compiler range checks. The run-time system uses interrupts 16 through 31. Interrupt number 16 is reserved for real arithmetic exceptions, and interrupt number 17 is reserved for stack overflow and case out-of-range checks.

You can use other interrupt numbers for your own procedures. If you intend to override the default procedures provided for the above interrupt numbers, you must use the above interrupt numbers for those procedures.

An interrupt occurs when the CPU receives a signal on its "maskable interrupt" pin from some peripheral device. The CPU responds, however, only if interrupts are enabled. The "main program prologue" (code inserted by the compiler at the beginning of the main program) does not alter the state of interrupts. (This differs from the PL/M-86 Compiler.) In other words, when you execute a Pascal program, the interrupts are in the same state as they were before execution.

If interrupts are enabled, the following actions take place:

1. The CPU issues an "acknowledge interrupt" signal and waits for the interrupting device to send an interrupt number.
2. The CPU flag registers are placed on the stack (occupying two bytes of stack storage).
3. Interrupts are disabled by clearing the IF flag.
4. Single stepping is disabled by clearing the TF flag.
5. The CPU activates the interrupt procedure corresponding to the interrupt number sent by the interrupting device.

You can specify Pascal-86 procedures as interrupt procedures by using the INTERRUPT control (10.3.11). Using the INTERRUPT control or the SETINTERRUPT built-in procedure (8.9.1), you can assign an interrupt number to each interrupt procedure. These interrupt numbers form an *interrupt vector*, which is an absolutely-located array of entries beginning at location 0. Thus, the *n*th entry is at location 4 times *n*, and contains the address of the interrupt procedure associated with interrupt number *n*. Each entry is a four-byte value containing a segment address and an offset (i.e., a long pointer).

The CPU uses the interrupt vector entry to make a long indirect call to activate the appropriate procedure. At this point, the current code segment address (CS register contents) and instruction offset (IP register contents) are saved on the stack.

An interrupt procedure can also be activated by a procedure statement. If you intend to use a procedure statement in a module to activate an external interrupt procedure in another module, you *must* use the INTERRUPT control in the PUBLIC declarations of the external interrupt procedure (in the interface specifications of both modules); otherwise, an error will occur during the linking process (LINK86). A procedure statement will activate the interrupt procedure as if an interrupt occurred, and the interrupt status will be altered as if an interrupt occurred.

If an interrupt procedure terminates normally (other than by a GOTO statement), the interrupt mechanism and registers are reset to the condition that existed prior to the activation of the procedure.

Figure K-1 shows the stack layout at the point where the procedure is activated.

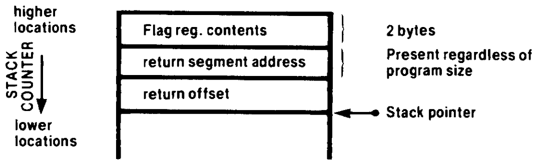


Figure K-1. 8086 Stack Layout When Interrupt Procedure Gains Control

121539-41

K.1.1 Interrupt Procedure Preface and Epilogue

At the beginning of each interrupt procedure, before the usual procedure prologue inserted by the compiler, the compiler inserts an *interrupt procedure preface* that performs the following actions:

1. Push the ES register contents onto the stack.
2. Push the DS register contents onto the stack.
3. Push the AX register contents onto the stack.
4. Push the CX register contents onto the stack.
5. Push the DX register contents onto the stack.
6. Push the BX register contents onto the stack.
7. Push the SI register contents onto the stack.
8. Push the DI register contents onto the stack.
9. Load the DS register with a new data segment address taken from the current code segment (i.e., the segment containing the interrupt procedure).

Figure K-2 shows the stack layout at the point where the procedure prologue starts.

Figure K-3 shows the stack layout after the procedure prologue is executed and the code compiled from the interrupt procedure body starts executing.

When the interrupt procedure body finishes, the *interrupt procedure epilogue* continues with the following steps:

10. Pop the stack into the DI register.
11. Pop the stack into the SI register.
12. Pop the stack into the BX register.
13. Pop the stack into the DX register.
14. Pop the stack into the CX register.

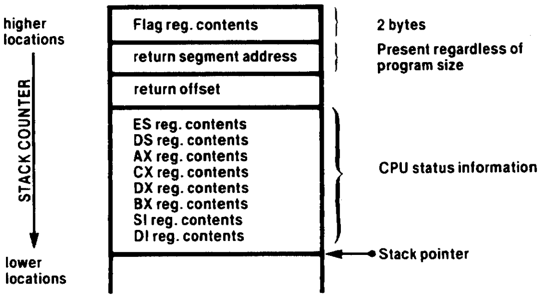


Figure K-2. 8086 Stack Layout after Interrupt Procedure Preface and before Procedure Prologue 121539-42

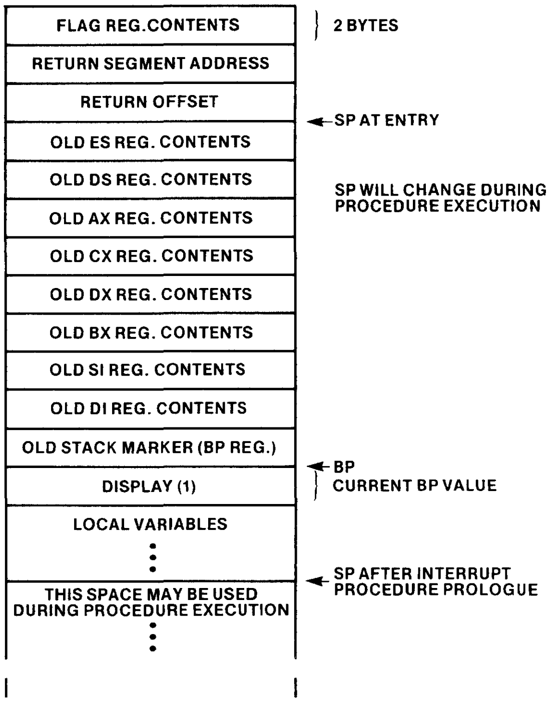


Figure K-3. 8086 Stack Layout during Execution of Interrupt Procedure Body

121539-43

15. Pop the stack into the AX register.
16. Pop the stack into the DS register.
17. Pop the stack into the ES register.
18. Execute an IRET instruction to return from the interrupt procedure. This restores the IP, CS, and flag register contents from the stack.

At this point the stack is restored to the state it was in before the interrupt occurred, and processing continues normally.

The INTERRUPT compiler control gives you the opportunity to associate an interrupt number with an interrupt procedure during compile time. You can also declare procedures as interrupt procedures without associating them to interrupt numbers, and create the interrupt vector at a later time, to be linked to the program.

Similarly, you could have a library of interrupt procedures that are not yet associated with an interrupt vector. Any program could then have any of these procedures linked in, with a separately created interrupt vector.

NOTE

An interrupt procedure that uses any of the built-in functions EXP, LN, SIN, COS, TAN, ARCSIN, ARCCOS, ARCTAN, TRUNC, ROUND, LTRUNC, or LROUND (functions in the CEL87.LIB run-time library) must allocate 50 bytes of 8086 stack space (200 bytes if the 8087 emulator is used) for each level of recursion. The 8087 chip or emulator cannot be accessed from both a Pascal interrupt procedure and a concurrent Pascal program unless the interrupt procedure saves the status of the 8087.

K.2 Pascal Run-Time Storage Management

Pascal run-time storage management, required by the predefined procedures NEW and DISPOSE, is provided by two memory management systems. The Pascal compiler automatically determines which system to use, depending on what model of segmentation the program is compiled under. Programs compiled with the SMALL control (with —CONST IN DATA—) use 16-bit pointers and require that the storage allocated be in DGROUP. All other models use 32- bit pointers and have no restriction on what segment or group they point to. Both systems may be used in the same program if it contains both SMALL and non-SMALL subsystems.

K.2.1 Memory Managers

The SMALL Model

The SMALL memory manager allocates storage from an area called the heap (by default, the entire MEMORY segment in DGROUP). If your program uses the NEW procedure, the memory manager will create this segment with a size of 4096 bytes. You may adjust the size of MEMORY at link time by using the LINK86 controls MEMPOOL or SEGSIZE. For example, using SEGSIZE(MEMORY(+0, 0FFFFH)), the MEMORY segment will expand to fill DGROUP, whose maximum size is 64K. You can reduce the size of MEMORY below 4096 bytes, but LINK86 will generate a warning.

The memory manager determines the size of MEMORY from the size of DGROUP, which it gets by calling the UDI primitive DQGETSIZE. This is possible because the relocatable loader uses DQALLOCATE to get the storage for segments and groups. However, to use the SMALL memory manager with an absolute program (one located with LOC86), you must indicate the size of the heap to the memory manager. If you fail to do so, an exception occurs.

You may override the default location and/or the size of the SMALL heap by providing your own version of the Logical Record System routine TQGETSMALLHEAP. The memory manager calls this routine the first time an allocation request is made to determine the location and size of the SMALL heap.

The TQGETSMALLHEAP primitive is called as a procedure and does not return a value, hence it cannot be a typed procedure. It receives three VAR WORD arguments that are assigned the following values: the offset in DGROUP of the start of the heap, the size of the heap (in bytes), and an exception code if it occurs. (The run-time system does not require non-zero exception codes.)

The following is a sample PL/M-86 procedure heading for TQGETSMALLHEAP:

```
TQGETSMALLHEAP: PROCEDURE(OFFSET_PTR,
                          SIZE_PTR,
                          EXCEPTION_PTR) PUBLIC;
                DECLARE OFFSET_POINTER POINTER,
                        SIZE_PTR POINTER,
                        EXCEPTION_PTR POINTER;
                END TQGETSMALLHEAP;
```

The heap returned by TQGETSMALLHEAP must be entirely within DGROUP, and must be at least 16 bytes long.

Refer to the *Run-Time Support Manual for iAPX 86,88 Applications* for more information on the SMALL heap.

The LARGE Model

The LARGE memory manager is used for all subsystems that are not SMALL with (—CONST IN DATA—). This memory manager gets storage from the operating system by using the LRS routine TQALLOCATE. Memory is requested from the operating system in 1024-byte "pages." All requests for 251 bytes or less are satisfied from these pages. If no page contains a large enough block of free storage, another page is requested from the operating system. When all of the storage in a page has been released by the Pascal program, the page is returned to the operating system through the LRS procedure TQFREE. Requests for more than 251 bytes are passed directly to TQALLOCATE.

K.2.2 Reentrancy

The LARGE model is designed so that it can be used in a multi-tasking environment. All of the non-reentrant code in the memory manager is in the run-time library P86RN0.LIB. Each task must contain its own copy of these library routines; simply link P86RN0.LIB to each task before they are linked together. If the individual tasks are not linked with the PURGE option before being linked together, LINK86 will generate warnings about duplicate publics. Refer to the *Run-Time Support Manual for iAPX 86,88 Applications*, Order Number 121776, to determine how each task interacts with its heap. The routines in P86RN1.LIB, P86RN2.LIB, and P86RN3.LIB are reentrant and may be shared among concurrently executing tasks.

The SMALL memory manager is not reentrant, since all SMALL subsystems in a job share the same heap in the MEMORY segment of DGROUP. If you want more than one task to use the SMALL memory manager, you can supply your own version of TQGETSMALLHEAP that returns a different heap area to each task. You could also provide a synchronized Pascal program, ensuring that no two SMALL model tasks are executing a call to a NEW or DISPOSE procedure at the same time.

K.2.3 Replacing the Memory Manager

For some applications, such as interfacing to certain operating system routines, it is necessary for a pointer to have an offset part of 0. This type of pointer may be translated to a two-byte SELECTOR (as in PL/M and RMX-86). Note that routines using pointers this way must be written in assembly language or PL/M, since Pascal does not support tokens. You can also generate pointers with an offset of zero using the NEW procedure. Replace the normal memory manager with routines that pass all allocation and release requests directly to the UDI routines DQALLOCATE and DQFREE (or to the LRS routines TQALLOCATE and TQFREE). The Pascal compiler generates code to call the following routines for the LARGE memory manager:

NEW

Entry point:	PQ_310	(FAR)	
Parameters:	AX		—size of area to allocate
Returns:	ES:BX		—pointer to area allocated

DISPOSE

Entry point:	PG_320	(FAR)	
Parameters:	Stack1:Stack2		—pointer to area to dispose
	AX		—size of area

Stack2 is the WORD on top of the stack, and Stack1 is the second WORD from the top. The routines you write to replace the normal memory manager need to be linked in before the first Pascal library P86RN0.LIB. This method cannot be used for SMALL programs, since the allocated storage must be in DGROUP.

K.2.4 Memory Usage Summary

The following values help determine the actual amount of memory that will be used by the program:

SMALL Memory Manager:

Overhead (per allocation)	—	2 bytes (If necessary, another byte is used to make the length of the allocated storage block even.)
Global overhead	—	6 bytes
Minimum allocation	—	6 bytes (without padding)

LARGE Memory Manager (request ≤ 251 bytes):

Overhead (per allocation)		2 bytes (If necessary, another byte is used to make the length of the allocated storage block even.)
Minimum allocation	—	6 bytes (without padding)
Size of page requested from operating system	—	1024 bytes
Overhead (per page)	—	6 bytes

LARGE Memory Manager (request > 251 bytes):
Allocation requests are made to the operating system.

The run-time system initialization routine (TQ-001) allocates an eighteen-byte data area for run-time system use.

The Pascal I/O system also dynamically requests storage from the operating system. When a file is opened, a 48-byte file descriptor is allocated by the LRS procedure TQFILEDESCRIPTOR. The default TQFILEDESCRIPTOR uses DQALLOCATE to allocate this storage. A file is opened when the first RESET or REWRITE is performed on it (including the implicit RESET and REWRITE performed on the files INPUT and OUTPUT).

The default LRS also uses DQALLOCATE to allocate a 1054-byte buffer for each disk file that is opened. This buffer is returned to the operating system when the file is closed.

The Pascal I/O system also allocates an 86-byte line buffer when a REWRITE is performed on a file of type TEXT or FILE OF CHAR. This buffer is allocated above the level of the LRS by using the LRS routine TQALLOCATE; its purpose is to improve the efficiency of output to a console. The buffer is released using TQFREE when a RESET is performed on the file or when the file is closed. The I/O system will not buffer the output to files of type TEXT or FILE OF CHAR if TQALLOCATE returns a non-zero exception code. This allows you to write your own LRS without providing TQALLOCATE, yet still be able to use these files.

K.2.5 Allocate a New Memory Block

The run-time system calls this LRS allocation primitive whenever a Pascal-86 program uses the NEW procedure for dynamic memory allocation (to obtain memory blocks from the heap). The run-time system sends two argument values to this primitive: a WORD containing the number of contiguous bytes to be allocated, and a POINTER to a location where this primitive should return a status WORD containing an exception condition (if an exception occurs). There are no exception codes required by the run-time system.

The procedure you supply for this primitive must return the address of the first segment boundary of the memory block allocated for use by the Pascal-86 program.

The following is a sample PL/M-86 typed procedure heading for the TQALLOCATE primitive:

```
TQALLOCATE:     PROCEDURE(SIZE,STATUS_PTR) SELECTOR PUBLIC;
                DECLARE SIZE WORD;
                DECLARE STATUS_PTR POINTER;
                DECLARE MEM_SEL SELECTOR;
                .
                .
                .
                RETURN MEM_SEL;
                END TQALLOCATE;
```

The argument sent to the SIZE word parameter contains the number of contiguous bytes of memory that the Pascal-86 program wants to obtain from the heap. The argument sent to the STATUS_PTR pointer parameter is the location where the TQALLOCATE procedure should return a status WORD with an exception code if an exception occurs (an exception code is not required). In addition, the TQALLOCATE procedure must return MEM_SEL, a SELECTOR containing the address of the first segment boundary of the obtained block of memory.

Your TQALLOCATE primitive can allocate memory with a byte granularity of sixteen, since the run-time system has a memory manager to manage large blocks.

K.2.6 Free a Previously Allocated Memory Block

The run-time system calls this LRS primitive to free the block of memory obtained by the TQALLOCATE primitive described in the previous section. When a Pascal-86 program uses the DISPOSE procedure to dispose of the memory obtained from the heap, the run-time system calls this TQFREE primitive to return the memory to the heap.

The TQFREE primitive is called as a procedure, and it does not return a value (hence it cannot be a typed procedure). The primitive receives a SELECTOR argument containing the address of the first segment boundary of a memory block obtained via the TQALLOCATE primitive described in the previous section. The primitive also receives a pointer to a location where the primitive should place a status WORD with an exception code if an exception occurs (no exception codes are required for the run-time system).

The following is a sample PL/M-86 procedure heading for the TQFREE primitive:

```
TQFREE:         PROCEDURE(MEM_SEL, STATUS_PTR) PUBLIC;
                DECLARE MEM_SEL SELECTOR;
                DECLARE STATUS_PTR POINTER;
                .
                .
                .
                END TQFREE;
```

K.3 Logical Record Interface

Intel provides a *logical record interface* that allows you to hook an operating environment or application-dependent device drivers to the run-time support system for Pascal-86.

The diagram in figure K-4 shows typical execution paths for Pascal-86·programs. The default path uses the interface libraries to the Series III operating system. If you intend to use this path, you do not need the information in this appendix.

Another path shows a deviation at level (2), where you can interface your own *logical record system* to the logical record interface. Your logical record system would be a collection of primitives, file drivers, and/or device drivers, or an interface library to your own operating system.

You can also use Intel's logical record system at level (2) and hook your own operating system to it at level (3), as shown. To hook your own operating system to Intel's logical record system and run-time system, you have to supply primitives that have the same names as the primitives used to interface the Series III operating system with the Pascal-86 run-time system and logical record system. A list of the names appears at the end of this appendix.

Real arithmetic support (via the 8087 or emulator) is not depicted in this diagram, but you must use either the 8087 processor and its library (8087.LIB), the emulator and its library (E8087, E8087.LIB), or 87NULL.LIB for floating-point programs, as described in Chapter 12.

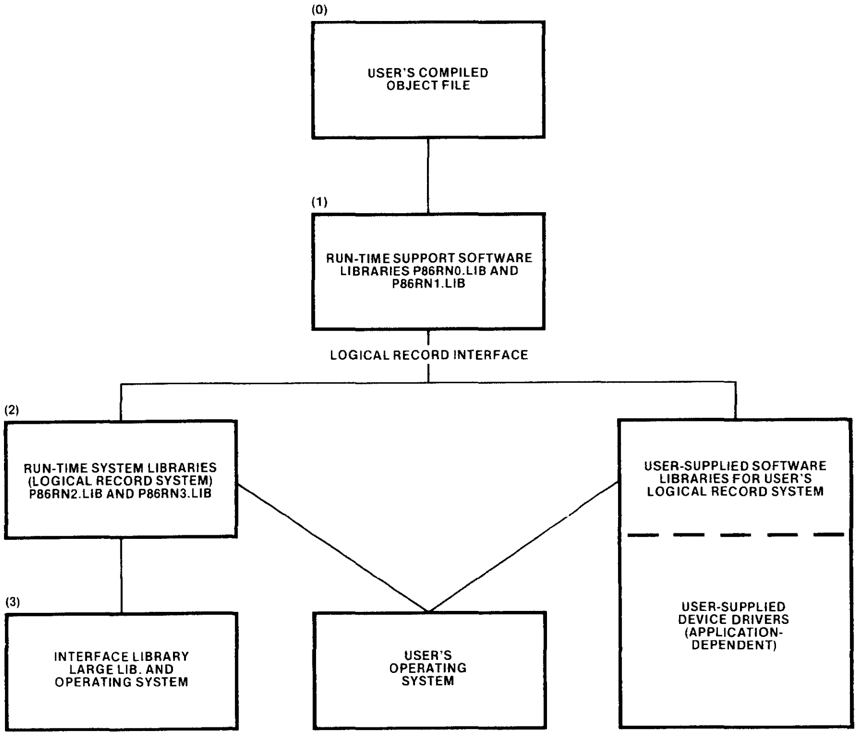

(0)
USER'S COMPILED OBJECT FILE
(1)
RUN-TIME SUPPORT SOFTWARE LIBRARIES P86RN0.LIB AND P86RN1.LIB
LOGICAL RECORD INTERFACE
(2)
RUN-TIME SYSTEM LIBRARIES (LOGICAL RECORD SYSTEM) P86RN2.LIB AND P86RN3.LIB
USER-SUPPLIED SOFTWARE LIBRARIES FOR USER'S LOGICAL RECORD SYSTEM
USER-SUPPLIED DEVICE DRIVERS (APPLICATION-DEPENDENT)
(3)
INTERFACE LIBRARY LARGE LIB. AND OPERATING SYSTEM
USER'S OPERATING SYSTEM

Figure K-4. Execution Paths for Pascal-86 Programs 121539-44

The Pascal-86 run-time support system at level (1) in figure K-4 considers a file to be a series of logical records. A logical record contains exactly one object of a non-text file, or one line of a text file.

A user-defined logical record system that interfaces to the run-time system at level (2) in figure K-4 must be able to store a logical record in a physical record on a storage medium (such as a disk), and must be able to delimit such records. The run-time system uses the logical record interface primitives to call the user-defined logical record system, which formats logical file records onto physical media either explicitly through a set of device drivers, or implicitly through a set of file drivers and an operating system.

The logical record interface consists of a set of calling sequences to primitives; you substitute your own procedures or functions using the names of these primitives, which in turn use your logical record system routines. The primitives can be written in PL/M-86 (conforming to the LARGE model of segmentation), Pascal-86, or ASM86. The run-time system requires that some of these primitives return a status word to indicate a status or exception condition, which directs the functionality of the run-time system.

If you code your logical record system in Pascal-86, in some cases you will have to supply your own operating system primitives to provide the run-time support for your logical record system (since the supplied run-time system relies on the default logical record system, or your logical record system). If you code them in PL/M-86 or in ASM86, you won't need run-time support for your logical record system; therefore, we show sample procedure headings in PL/M-86.

The following descriptions show sample PL/M-86 procedure headings, with parameter lists for each procedure you should supply. You use these parameters to pick up arguments sent to these primitives from the run-time system. In Pascal-86, these primitives must be functions in order to return a status value; in PL/M-86, they must be typed procedures.

These parameters are described in terms of PL/M-86 data types. For a table of Pascal-86 data types and their corresponding PL/M-86 types, see Appendix J of this manual. Appendix J also describes the universal calling sequence used by Pascal-86 and other iAPX 86,88 family languages, so that modules written in one of these languages can easily call modules written in another.

The logical record system you supply must handle file input/output, file preconnection, exception conditions, and memory (heap) allocation. You can, however, use Intel's logical record system (default primitives) found in library P86RN2.LIB, and still provide your own device drivers by supplying your own TQDEVICE primitive (K.3.2), as long as you link in your TQDEVICE primitive and device drivers before linking in the P86RN2.LIB library (in order to override default external references).

The logical record system you supply must set up two data structures: a *file/device descriptor* used by both the run-time system and your logical record system to hold status information about each file or device (one descriptor for each file or device), and a *file/device driver table* defined by your logical record system that provides the addresses of your file/device driver routines. The P86RN0.LIB and P86RN1.LIB libraries do not perform any buffering. Depending on your execution environment, you may want to implement buffering in your logical record system to improve execution speed.

After writing your logical record system primitives, you must follow the linking conventions described in K.3.7 to link them to your Pascal-86 program modules.

K.3.1 Setting Up the File/Device Descriptor

The run-time system calls the TQFILEDESCRIPTOR primitive after a file preconnection, or before a call to open a file if there is no file preconnection, to set up a block of memory (called a file/device descriptor) used to store attributes of a file or device. This block of memory is reserved for the file or device attributes until the file is closed. After the file is closed, the file/device descriptor block is used again by the run-time system for other files. Replace this primitive only if you are not using a UDI operating system.

The file/device descriptor must be 48 bytes, and must start on a segment boundary (i.e., it must begin at a location that is a multiple of 16 bytes). The initial (least significant) 16 bytes of the file descriptor are for your file attributes—the run-time system does not disturb this area if you are also providing your own device drivers (discussed in the next section). The run-time system uses the last (most significant) 32 bytes to contain file management information.

The run-time system expects the TQFILEDESCRIPTOR primitive to have one parameter and to return one value, and it must be declared to be public. The TQFILEDESCRIPTOR primitive returns a status WORD value that holds an exception condition code if an exception occurs, but none are required by the run-time system (the exception codes required for the other primitives are listed in K.3.6). The run-time system calls TQFILEDESCRIPTOR in an assignment statement:

```
STATUS=TQFILEDESCRIPTOR(FDSEGPTR)
```

The following is a sample PL/M-86 typed procedure heading for the TQFILEDESCRIPTOR primitive:

```
TQFILEDESCRIPTOR: PROCEDURE(FD_SEG_PTR) WORD PUBLIC;
                  DECLARE(FD_SEG_PTR) POINTER;
                  .
                  .
                  .
                  END TQFILEDESCRIPTOR;
```

The primitive receives an argument for the POINTER parameter FD_SEG_PTR, which contains the address for a SELECTOR that contains the selector of the file descriptor. This selector word is sent as an argument to the parameter FD_SEL used in every device driver.

Since TQFILEDESCRIPTOR is a PL/M-86 typed procedure, it must use a RETURN to return the WORD status value. If you intend to code it in Pascal-86 as a function, the function must assign the status value to the name of the function (TQFILEDESCRIPTOR).

K.3.2 Connecting File/Device Drivers

Routines that actually transfer data and communicate with external files or devices are called *file/device drivers*. The run-time system provides two default drivers in P86RN2.LIB and P86RN3.LIB: one for text files, and one for non-text files. The run-time system assumes that ten actions can be performed for each file: opening the file, closing the file, reading, writing, moving forward, marking the end of a record, rewinding, seeking, marking the end of a file, and getting information about a file.

The Logical Record System (LRS) interface has been modified to accommodate random I/O. If you have written your own LRS interfaces, and if you plan to use the random access I/O capability, you will be affected by this change. (If you use the random access feature and Intel libraries, you will be able to use this I/O capability immediately.)

The random access features will use two UDI primitives, DQFILEINFO and DQSEEK, which previously were not used by Pascal-86. Thus, if you have written your own UDI interface but did not implement all the UDI primitives, you must add these before using random access I/O.

Two new entries have been added to the device driver table: file information and an entry reserved for future use. The new device driver table is shown in figure K-5.

You can replace or supplement these drivers with your own drivers that communicate with your operating environment by formatting your driver routines to correspond to the calling sequence and parameter lists that the run-time system expects to find. The run-time system uses the default drivers unless you explicitly create your own TQDEVICE primitive as part of your logical record system. If you supply your own TQDEVICE, you can associate a file with your own set of drivers. To supply your own TQDEVICE primitive, use the name TQDEVICE, declare it to be public, and follow the module linking conventions described in K.3.7.

Before opening any file, the run-time system calls TQDEVICE and sends arguments for three parameters: a POINTER to the physical filename, a BYTE containing the length of the filename, and a POINTER to the base of a table containing the addresses of the drivers. TQDEVICE must return a WORD status value containing a zero if it

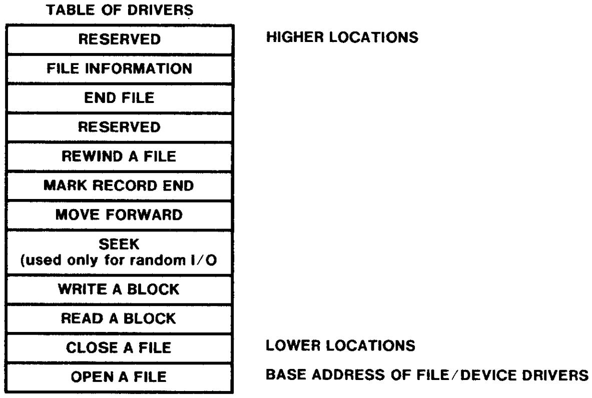


Figure K-5. Table of Addresses for File/Device Drivers 121539-26 is successful, or an exception code if an exception occurs. The following is a sample PL/M-86 typed procedure heading for the TQDEVICE primitive:

```
TQDEVICE:  PROCEDURE(NAME_PTR,
                     NAME_LENGTH,
                     DRIVER_TABLE_PTR) WORD PUBLIC;
           DECLARE DRIVER_BASE BASED
                     DRIVER_TABLE_PTR POINTER;
           /* Assign the new table base to
              DRIVER_BASE to override the
              default assignment */
           DECLARE (NAME_PTR,DRIVER_TABLE_PTR) POINTER;
           DECLARE NAME_LENGTH BYTE;
           .
           .
           .
           END TQDEVICE;
```

If you wish to provide device drivers for some but not all files, your TQDEVICE primitive should examine the filename by using the pointer NAME_PTR (and the length byte NAME_LENGTH) to find the filename. For files that need your new set of device drivers, your primitive would alter the value pointed to by DRIVER-_TABLE_PTR, in order to use the new base address of your table of driver addresses (as described below). For files that need the default device drivers supplied for the Intel Series III operating system, your primitive would *not* alter the value pointed to by DRIVER_TABLE_PTR, and the base address for the default table would be used.

To supply your own set of device drivers, you must write the drivers and place the address of each driver in a file/device driver table, as shown in figure K-5. Each driver is described following this table. Your version of TQDEVICE would use the base address of this table (the address of the driver to open a file) as the value pointed to by DRIVER_TABLE_PTR for any file that needs this table of drivers. You can easily create alternatives in your TQDEVICE for certain files that need different kinds of drivers.

Since the run-time system calls these drivers by using the addresses in this table, the drivers themselves do not have to have special names. These drivers are described below.

Open a File

Before the run-time system performs any input/output, it calls a driver to open a file by using the address in the file/device driver table shown in figure K-5. To open a file, the file must be already rewound; i.e., the "file pointer" must be pointing to the beginning of the file. The following is a sample PL/M-86 typed procedure heading for the OPEN_FILE.driver:

```
OPEN_FILE: PROCEDURE(FD_SEL, NAME_PTR,
                NAME_LENGTH, ATTRIBUTE,
                REC_LENGTH) WORD PUBLIC;
           DECLARE (FD_SEL, ATTRIBUTE, REC_LENGTH)
                SELECTOR;
           DECLARE NAME_PTR POINTER;
           DECLARE NAME_LENGTH BYTE;
           .
           .
           .
           END OPEN_FILE;
```

This driver must be a typed procedure that returns a WORD value containing either an exception code if an exception occurs, or zero if the open action is successful. The run-time system obtains the arguments for the NAME_PTR pointer and the NAME_LENGTH byte from either the Pascal-86 program (the object module) or the TQGETPRECON primitive (described in K.3.4). If the argument for NAME_LENGTH equals zero, the driver should assume that it is a scratch file to be deleted after the close file operation.

The argument for the FD_SEL selector parameter comes from the TQFILEDESCRIPTOR primitive described in K.3.1. The argument for the REC_LENGTH word parameter is the record length to be associated with the file, supplied by the Pascal-86 program (the object module); a value of zero indicates that the record length is variable, as in a text file.

The argument for the ATTRIBUTE word parameter contains information and file attributes that your OPEN_FILE driver can use or ignore, depending on whether the information and attributes are relevant for the device you are associating with the file. The bits of this WORD are defined as follows:

Bits 0, 1—reserved, set to 00.

Bit 2—reserved, set to 0.

Bit 3—form of file:

0—non-text file. The file will contain binary data, not character data.

1—text file. The file will contain character data. The run-time system will not perform run-time checks on input or output.

Bit 4—reserved.

Bit 5—reserved, set to 0.

Bit 6—interactive file:

0—not interactive file.

1—possibly interactive file. The device should be treated as an interactive console (indicating a file of type CHAR).

Bit 7—reserved, set to 0.

Bits 8 and 9—mode of file:

0—destructive write only. The file can only be written to (as after a REWRITE).

01—read only. The file can only be read (as after a RESET).

10—reserved

11—update. The file can be written to.

The remaining bits are reserved.

Close a File

The run-time system calls this driver to close a file by using the address provided in the file/device driver table shown in figure K-5. The run-time system reuses the file/device descriptor for the next open file operation; that is, it will avoid calling the TQFILEDESCRIPTOR primitive. This driver returns a status WORD with an exception code if an exception condition occurs (none are required by the run-time system).

The following is a sample PL/M-86 typed procedure heading for the CLOSE_FILE driver:

```
CLOSE_FILE:     PROCEDURE(FD_SEL, DISPOSE) WORD PUBLIC;
                DECLARE FD_SEL SELECTOR;
                DECLARE DISPOSE BYTE;
                .
                .
                .
                END CLOSE_FILE;
```

The argument for the FD_SEL selector parameter is the same as the argument in the OPEN_FILE driver, described in the previous section. The argument for the DISPOSE byte parameter can be ignored by this driver.

Read a Block from a File

The run-time system calls this driver to read a block of data from a file. This driver must return a status WORD containing an exception code if the end of a record or the end of the file condition is reached. These codes are listed in K.3.6. The following is a sample PL/M-86 typed procedure heading for the READ_BLOCK driver:

```
READ_BLOCK:     PROCEDURE(FD_SEL, BUFFER,
                        COUNT, ACTUAL_PTR) WORD PUBLIC;
                DECLARE (FD_SEL, COUNT) SELECTOR;
                DECLARE (BUFFER,ACTUAL_PTR) POINTER;
                .
                .
                .
                END READ_BLOCK;
```

The argument for FD_SEL comes from the TQFILEDESCRIPTOR primitive described in K.3.1. The other arguments are sent from the run-time system. The argument for the BUFFER pointer parameter is the base address of an area where this driver must store the block of data read. The argument for the COUNT word parameter is the length in bytes of the data item to be read. The driver should store the actual number of bytes read in the WORD addressed by ACTUAL_PTR.

For files that have fixed-length records, the run-time system never tries to read more bytes than the fixed number defined for the record. However, this read block driver must be able to recognize the end of a record, and be able to position the "file pointer" to the beginning of the next record for each read operation. This record delimiter might be the carriage return/line feed (CR/LF) combination (for text files) or a gap between records on a tape drive.

Write a Block to a File

The run-time system calls this driver to write blocks of data to a file. The run-time system calls this driver by using the address found in the file/device driver table shown in figure K-5. There might be more than one call to this driver to write an entire record, and this driver must handle any buffering. After calling this driver, the run-time system calls the "mark end of record" driver to mark the end of a record, even if the records are fixed in length.

The following is a sample PL/M-86 typed procedure heading for the WRITE_BLOCK driver:

```
WRITE_BLOCK:   PROCEDURE(FD_SEL, BUFFER,
                         COUNT) WORD PUBLIC;
               DECLARE (COUNT) WORD;
               DECLARE (FB_SEL) SELECTOR;
               DECLARE BUFFER POINTER;
               .
               .
               .
               END WRITE_BLOCK;
```

This driver returns a status WORD that contains the exception code of an exception condition if one occurs (none are required by the run-time system). The argument for the FD_SEL selector parameter comes from the TQFILEDESCRIPTOR primitive described in K.3.1. This driver uses the argument for the BUFFER pointer parameter as an address to find the data to be written, and it uses the argument for the COUNT word parameter as the length of the data block to be written.

Seek In A File

This routine is called for files opened for direct access to position the file pointer before a read or write operation. The run-time system calls this routine by using the address found in the table in figure K-5. The following is a sample PL/M-86 typed procedure heading for the SEEK driver.

```
SEEK: PROCEDURE (FD, MODE, HIGH$OFFSET, LOW$OFFSET) WORD
           PUBLIC REENTRANT;
      DECLARE FD           SELECTOR,
              MODE         BYTE,
              LOW$OFFSET   WORD,
              HIGH$OFFSET  WORD;
              .
              .
              .
      END SEEK;
```

where

FD identifies the file descriptor for the file to be affected.

LOW$OFFSET
HIGH$OFFSET together form a four-byte (DWORD) unsigned integer (here called offset) that represents either a position in the file or the number of bytes to move the file position pointer, depending on the setting of mode.

MODE indicates the type of seek required. The values of mode are defined as:

0 — Seek to the record number specified in offset. Note that the first record of a file is record number 1.

1 — Move file pointer back by offset bytes within current record.

2 — Set file pointer to offset within current record.

3 — Move file pointer forward by offset bytes within current record.

4 — Move file pointer to end of file.

This is a typed procedure (function). The value of the procedure is a WORD that indicates the result of calling this procedure. The required exception code is E$OK and the RTNULL version will cause processing to halt. Modes 1 through 4 are not currently supported or required.

Move Forward to the End of the Record

The run-time system calls this driver to skip to the end of the record and prepare the file for the next sequential access. The run-time system uses the address in the file/device driver table shown in figure K-5 to call this driver. This driver is called whenever the processing of a record has finished, even if the record was only partially read.

The following is a sample PL/M-86 typed procedure heading for the MOVE_FORWARD driver:

```
MOVE_FORWARD:   PROCEDURE(FD_SEL) WORD PUBLIC;
                DECLARE FD_SEL SELECTOR;
                .
                .
                .
                END MOVE_FORWARD;
```

The argument for the FD_SEL selector parameter comes from the TQFILEDESCRIPTOR primitive described in K.3.1. This driver must return a WORD containing a zero for a successful move operation, or the exception code for an end of file if that condition occurs. The exception codes are listed in K.3.6.

Mark the End of a Record

The run-time system calls this driver every time output to a particular record is completed. Your driver should mark the file as appropriate for the device associated with the file. For a file with fixed-length records, the driver may either increment the record pointer or pad the rest of the record with a distinguishable character. A call to this driver implies that the program has terminated output to the record, and that the rest of the record is either undefined or defined by this driver.

The following is a sample PL/M-86 typed procedure heading for the END_RECORD driver:

```
END_RECORD:     PROCEDURE(FD_SEL) WORD PUBLIC;
                DECLARE FD_SEL SELECTOR;
                .
                .
                .
                END END_RECORD;
```

The argument for the FD_SEL selector parameter comes from the TQFILEDESCRIPTOR primitive described in K.3.1. This driver returns a status WORD that contains the exception code of an exception condition if one occurs (none are required by the run-time system).

Rewind a File

The run-time system calls this driver to rewind a file, and the driver rewinds by moving the "file pointer" to the beginning of the file. Nothing occurs for devices that cannot rewind. The driver returns a status WORD that contains the exception code of an exception condition if one occurs (none are required by the run-time system.) The following is a sample PL/M-86 typed procedure heading for the REWIND driver:

```
REWIND:        PROCEDURE(FD_SEL,MODE) WORD PUBLIC;
               DECLARE FD_SEL SELECTOR;
               DECLARE MODE BYTE;
               .
               .
               .
               END REWIND;
```

The argument for the FD_SEL selector parameter comes from the TQFILEDESCRIPTOR primitive described in K.3.1. The argument for the MODE byte parameter is the value specified in bits 8 and 9 of the ATTRIBUTE word parameter for the OPEN_FILE driver. When the file is rewound, the MODE byte defines subsequent access rights to the file.

Mark the End of a File

The run-time system calls this driver to mark the current position of the file pointer as the end of the file. The following is a sample PL/M-86 typed procedure heading for the ENDFILE driver.

```
ENDFILE:       PROCEDURE (FD) WORD PUBLIC REENTRANT;
               DECLARE FD SELECTOR;
               .
               .
               .
               END ENDFILE;
```

This driver returns a status WORD that contains the exception code of an exception condition if one occurs (none are required by the run-time system). The argument for the FD parameter is the file descriptor for the file to be affected. If data is beyond the location indicated by the current file pointer, that data is truncated.

Get File Information

The run-time system may obtain information about a file by using the address provided in the file/device driver table shown in figure K-5. Following is a sample PL/M-86 typed procedure heading for the FILE_INFORMATION driver.

```
FILE_INFORMATION:  PROCEDURE (FD, FILE_INFO_P) WORD;
                   DECLARE FD SELECTOR;
                   DECLARE FILE_INFO_P POINTER;
                   .
                   .
                   .
                   END FILE_INFORMATION;
```

where FD identifies the file descriptor for the file to be affected and FILE_INFO_P is a POINTER to an area of memory with the following format:

```
DECLARE FILE_INFO_STRUCTURE (
                    CURR_POS DWORD,
                    FILE_LEN DWORD,
                    RESERVED (2) WORD);
```

CURR_POS is the record number at which the file is currently positioned, FILE_LEN is the record number of the last record in the file, and RESERVED is a two-word field that is reserved for future use.

K.3.3 Initialize the Logical Record System

The run-time system calls this initialization primitive before calling any primitive in the logical record system. You supply your own initialization primitive to initialize your logical record system, initialize any devices or tables you need for your logical record system, initialize the preconnection table of logical files connected to physical files, and provide your own semaphore mechanisms so that the run-time system can protect critical regions of code. In short, the run-time system calls this primitive to allow you to provide whatever special mechanisms you need to operate the run-time system in your environment. Do not replace Intel's default initialization primitive unless you are supplying your own logical record system.

The following is a sample PL/M-86 typed procedure heading for the TQINITIALIZE primitive:

```
TQINITIALIZE:   PROCEDURE(LRI_DATA_PTR) WORD PUBLIC;
                DECLARE LRI_DATA_PTR POINTER;
                .
                .
                .
                END TQINITIALIZE;
```

The argument sent for the LRI_DATA_PTR pointer parameter is the address of a location where this primitive must store a WORD. This WORD must contain the root address of a linked list of logical and physical file names which make up your preconnection table. This WORD will be used as an argument to a parameter of the TQGETPRECON primitive described in the next section. This primitive also returns a status WORD that may contain an exception code if an exception occurs (the run-time system does not require an exception code).

K.3.4 Return Physical File Name for Preconnection

The run-time system calls this primitive to obtain the physical file name to be associated with a Pascal-86 logical name by means of a file preconnection specification on the command line that executes a Pascal-86 program. (See 12.4 for a description of preconnecting files on the execution command line.)

This primitive must return a BYTE value for the length of the physical file name, and it must store a·pointer to the physical file name at a location specified by an argument to the primitive. The following is a sample PL/M-86 typed procedure heading for the TQGETPRECON primitive:

```
TQGETPRECON:   PROCEDURE(UNIT,L_FILENAME_PTR,
                         L_FILENAME_LENGTH,
                         P_FILENAME_PTR,
                         PRECON_ROOT) BYTE PUBLIC;
               DECLARE (UNIT,L_FILENAME_LENGTH)
                         BYTE;
               DECLARE (L_FILENAME_PTR,
                         P_FILENAME_PTR) POINTER;
               DECLARE PRECON_ROOT WORD;
               DECLARE P_FILENAME_LENGTH BYTE;
               .
               .
               .
               RETURN P_FILENAME_LENGTH;
               END TQGETPRECON;
```

The argument sent for the UNIT byte parameter has no use in Pascal run-time preconnection. The argument sent for the L_FILENAME_PTR pointer parameter is the address of the location where the logical file name is stored. The argument sent for the L_FILENAME_LENGTH byte parameter is the length (in bytes) of the logical file name. Your primitive must use the pointer in L_FILENAME_PTR and the length in L_FILENAME_LENGTH to pick up the logical file name.

The argument sent for the P_FILENAME_PTR pointer parameter is the address of the location where your primitive must store a pointer to the physical file name to be associated with the logical file name. The physical file name's length is the return value of the function, so that the run-time system can pick up the physical file name. If your primitive returns a zero as the physical file name's length, then the run-time system will assume that a match for the logical file name was not found in the preconnection linked list. The root of the preconnection linked list is sent for the PRECON_ROOT word parameter—this argument comes from the initialization primitive (TQINITIALIZE) described in the previous section.

K.3.5 Exit from the Logical Record System

The run-time system calls this primitive to terminate the logical record system. You can supply your own primitive to perform any routine measures you need to close down your logical record system. The TQEXIT primitive must not return a value (hence it cannot be a typed procedure).

The following is a sample PL/M-86 procedure heading for the TQEXIT primitive:

```
TQEXIT:        PROCEDURE(TERMINATION_TYPE) PUBLIC;
               DECLARE TERMINATION_TYPE WORD;
               .
               .
               .
               END TQEXIT;
```

The run-time system sends either a zero or a one as an argument for the TERMINATION_TYPE word parameter. A zero means a normal termination, and a one means a termination as a result of an exception condition.

The run-time system uses a normal call to this primitive, and the primitive must not return.

K.3.6 Run-Time Exception Handling

The run-time system expects the operating environment (your operating system or the device driver system) to provide the address of the exception handling procedure. The run-time system calls this exception handler using this address.

To provide this address, you can implement two primitives in your logical record system: TQSETERH to establish the current exception handler, and TQGETERH to provide the address of the current exception handler to the run-time system.

The following is a sample PL/M-86 procedure heading for the TQSETERH primitive:

```
TQSETERH:         PROCEDURE (PROC_ADDR) PUBLIC;
                  DECLARE PROC_ADDR POINTER;
                  .
                  .
                  .
                  END TQSETERH;
```

The TQSETERH primitive must allocate a storage area to store the address of the current exception handler. The TQGETERH primitive must return a POINTER in the location specified by the run-time system so that the run-time system can call the current exception handler. The following is a sample PL/M-86 procedure heading for the TQGETERH primitive:

```
TQGETERH:         PROCEDURE (PROC_ADDR_PTR) PUBLIC;
                  DECLARE PROC_ADDR_PTR POINTER;
                  DECLARE PROC_ADDR BASED PROC_ADDR_PTR POINTER;
                  .
                  .
                  .
                  END TQGETERH;
```

The following exception condition is in operating environment exception returned by the default logical record system or your system to the run-time system:

Exception Code	Meaning
1501H	Command line preconnection facility has detected an invalid preconnection syntax.

The run-time system expects to receive the following status codes from some of the primitives and drivers in the default logical record system or your system:

Status Code	Meaning
15FFH	A read was attempted but the "file pointer" pointed to the end of the file.
15FEH	A read was attempted past the end of a record.

The exception code range 1520H to 15FFH is reserved for exceptions you can define for your logical record system.

K.3.7 Linking Conventions

To link your own set of logical record system primitives with the required library modules and Pascal-86 modules, specify on the linker's command line your logical record system module *after* specifying the other modules, so that references to the logical record system primitives that are in the run-time system libraries will be satisfied.

In addition, you must link in the null library RTNULL.LIB to resolve run-time system references to the default primitives you are not using. If, however, you are using some of the default primitives found in the default libraries P86RN2.LIB and/or P86RN3.LIB, and you link in these default libraries, then you do not need RTNULL.LIB.

If you are using the default libraries P86RN2.LIB and P86RN3.LIB (the default logical record system), yet you wish to supply some device driver primitives to override the default primitives, then link in your device driver primitives (and your version of the TQDEVICE primitive described in K.3.2) before linking in the default libraries P86RN2.LIB and P86RN3.LIB.

For more examples of linking modules and for more information about the linking and locating process, see Chapter 12 or your specific operating-system appendix.

K.3.8 Interfacing to the Default Logical Record System

To use Intel's logical record system (the libraries P86RN2.LIB and P86RN3.LIB) with your own operating system (other than an Intel operating system), you have to provide primitives with the same names as the following primitives, and link your set of primitives in place of the LARGE.LIB library used for the Series III. The names are:

DQ$ALLOCATE
DQ$FREE
DQ$GET$SIZE
DQ$TRAP$EXCEPTION
DQ$GET$EXCEPTION$HANDLER
DQ$EXIT
DQ$DELETE
DQ$ATTACH
DQ$OPEN
DQ$CREATE
DQ$CLOSE
DQ$DETACH
DQ$READ
DQ$WRITE
DQ$SPECIAL
DQ$SEEK
DQ$TRUNCATE
DQ$GET$CONNECTION$STATUS
DQ$GET$ARGUMENT
DQ$FILE$INFO

For details on the Intel versions of these primitives, see the *Intellec Series III Microcomputer Development System Programmer's Reference Manual*, Order Number 121618.

APPENDIX L COMPILER INVOCATION AND ADDITIONAL INFORMATION FOR SERIES III USERS

This appendix contains information that is specific to the Intellec Series III Microcomputer Development System. It covers the following areas:

- Compiler invocation and file usage
- Sample link, locate, and execute operations
- Examples of compiler control invocation lines
- Interrupt handling on the Series III
- Related publications

L.1 Compiler Operation

The Pascal-86 *compiler* is a program that translates your Pascal instructions into object code modules that can be linked and located for execution.

You create a Pascal program by typing instructions into a file using the CREDIT text editor, and submitting the file to the Pascal-86 compiler. The file you submit is called a *source file*, and the file containing the compiled program is called an *object file*. (The content of the object file is also known as *object code.*) In Pascal-86 you can compile *parts* of a program, and each separate compilation is known as an *object module*.

The following discussions assume that you have a Series III system up and running, and that you have a suitable copy of the Pascal-86 compiler. Chapter 1 of this manual leads you through a complete program development sequence using a sample Pascal program supplied with the compiler. Details on the operating system environment are provided in the *Intellec Series III Microcomputer Development System Console Operating Instructions*, Order Number 121609.

L.1.1 Invoking the Compiler

You invoke the Pascal-86 compiler by using the RUN command. The RUN command is used to load and execute any program specifically in the 8086 environment for the Series III system. The following is a sample compiler invocation:

```
-RUN PASC86 PROG1.SRC XREF<cr>
```

The name PASC86 is the name of the compiler as supplied, without the extension (i.e., the full name is PASC86.86, but you don't supply the .86 extension in the invocation line). PROG1.SRC is the name of the source file that contains the Pascal instructions. XREF is a primary control which tells the compiler to generate a cross-reference listing of source program identifiers (XREF is described in 10.3.23). The XREF control, like all other compiler controls, is optional for the invocation line.

The above example assumes that the compiler and the source program PROG1.SRC reside on drive 0 (:F0:). If PROG1.SRC is on drive 1, the invocation line is:

```
-RUN PASC86 :F1:PROG1.SRC XREF<cr>
```

The *invocation line* takes this general form:

```
RUN [:Fd:]PASC86 [:Fd]source[controls]
```

where

RUN is the name of the command to execute the compiler.

:F*d*: specifies which drive PASC86.86 and/or *source* resides on, if not on drive 0. The *source* file does not have to be on the same drive as the compiler.

PASC86 is the name you use for the compiler PASC86.86.

source is the name of the source file containing the Pascal program.

controls are optional primary or general compiler controls described in Chapter 10. You can have many controls in the invocation line with a space between each control, and you can extend the invocation line by using the ampersand (&) as a continuation character to replace a space.

< cr > stands for use of the RETURN key on the keyboard.

The following are some examples:

```
-RUN :F1:PASC86 :F1:MYPROG PRINT(:LP:) TITLE('TEST 24')<cr>
```

In this example, both PASC86.86 and MYPROG are on drive 1. PRINT and TITLE are compiler controls.

```
-RUN PASC86 :F1:KLUDGE.SRC NOPRINT<cr>
```

In this example, PASC86.86 is on drive 0, but KLUDGE.SRC, the source program, is on drive 1. NOPRINT is a compiler control that prevents all printed output (except error messages) usually generated by the compiler.

NOTE

The RUN command assigns the extension 86 to any filename you specify without an extension. You must specify the filename's extension if it is not 86. If you specify a filename that has no extension, specify a period (.) after the name in the RUN invocation line. For example, if you rename PASC86.86 to COMPIL, include a period after the name COMPIL (i.e., COMPIL.) when you invoke it using RUN. If you choose a new name with a new extension, specify both the new name and the new extension on the RUN invocation line.

L.1.2 Files Used by the Compiler

Input Files

You supply the Pascal source program name for *source* in the invocation line (see the previous section). You can also include other source files by using the INCLUDE control, as described in 10.3. These files must be standard ISIS-II disk files containing the text of Pascal instructions.

Output Files

By default, the compiler produces two output files, unless you use specific controls to suppress or redirect them: the *listing* file and the *object* file. Also by default, error messages appear at your console and in the listing file.

The listing file (sometimes called the PRINT file) contains a listing of the source program, plus any other printed output generated by the compiler as specified by the listing selection controls described in Chapter 10. The object file (sometimes called the object code file or object module) contains the actual code in object module format, which can eventually be executed (after you use the linking and locating facilities described in Chapter 13). These files are described in more detail in Chapter 12.

The listing file and the object file have the same name as the source file, except that the listing file has the extension LST, and the object file has the extension OBJ. The files are created if they do not exist, or overwritten if they do exist, and they appear on the same drive as the source file. You can optionally change the names and/or drives for the listing and object files by using the PRINT and OBJECT controls, respectively (described in 10.3).

For example, if you invoke the compiler using the line:

```
-RUN PASC86 :F1:MYPROG<cr>
```

the compiler creates (or overwrites) the file MYPROG.LST on drive 1 to contain the listing, and the file MYPROG.OBJ on drive 1 to contain the object module.

You can optionally direct certain sections of printed output to files other than the default listing file described above. In addition to using the PRINT control to specify another file as the listing file, you can specify a different file to receive error messages by using the ERRORPRINT control. Section 10.3 gives details on the use of these controls.

Work Files

The compiler creates and uses *work files* during its operation, and deletes them at the completion of compilation. These files are designated :WORK: files and they cannot conflict with your files.

The Series III operating system provides a mechanism to select the drive where work files can be temporarily stored. The default drive is drive 1 (:F1:), but you can select another drive using the RUN WORK command, as in this example:

```
-RUN WORK :F0:<cr>
```

This example selects drive 0 as the drive to hold work files.

L.1.3 Compiler Messages

When you invoke the compiler, it displays the sign-on message:

```
Series-III Pascal-86, Vx.y
Copyright 1981, 1983, 1984 Intel Corporation
```

where

x is the version number of the compiler.

y is the change number within the version.

As the compilation proceeds through its phases, the compiler displays messages that trace the compilation. As each phase starts, the name of the phase is displayed; when the phase terminates, the numeric parameter (if any) and comma are added. The name of the phase is prefixed by NO if the phase was not executed. The name of each phase (except for PARSE and ANALYZE) is the same as the control name that defines the phase.

These messages take the form:

```
PARSE(n), ANALYZE(n), [NO]XREF, [NO]OBJECT
```

where

n is the number of errors detected during execution of that particular phase.

The output files controlled by the PRINT and ERRORPRINT controls may be directed to the console (:CO:), in which case the compilation trace messages are interrupted with END clauses to show when a phase ends.

When a compilation is finished, the compiler terminates with the message:

```
Compilation of module verdict, n Errors[s] Detected.
End of Pascal-86 Compilation.
```

where

module is the name of the source module,

verdict is either ABORTED or COMPLETED, and

n is the total number of errors detected during the compilation.

L.2 Linking, Locating, and Executing on the Series III

L.2.1 Sample Link Operations

The following link operation takes two object modules, MYMOD1.OBJ and MYMOD2.OBJ, links them together, then links in the Pascal run-time libraries to form the output module MYPROG.86. To extend the LINK86 command to the next line without transmitting the command, type the ampersand (&) character before the RETURN key, and continue typing the command on the next line (do not type the ampersand character between letters of a file name). The continued line will start with two asterisks (**):

```
-RUN LINK86 MYMOD1.OBJ, MYMOD2.OBJ, P86RN0.LIB, &<cr>
**P86RN1.LIB, P86RN2.LIB, P86RN3.LIB, 87NULL.LIB, &<cr>
**LARGE.LIB TO MYPROG.86 BIND <cr>
```

The linker first reads MYMOD1.OBJ and MYMOD2.OBJ for external references and resolves those references between them. Then, the linker attempts to resolve any more external references in the modules by looking at the public symbols in the libraries P86RN0.LIB, P86RN1.LIB, P86RN2.LIB, P86RN3.LIB, 87NULL.LIB, and LARGE.LIB. Use of 87NULL.LIB implies that the modules do not perform real arithmetic. The final output module is MYPROG.86, which can then be loaded and executed on the Series III.

If the modules MYMOD1.OBJ and MYMOD2.OBJ do perform real arithmetic, then they must be linked with either the 8087 Numeric Data Processor or the 8087 Emulator. If you are using the emulator, the LINK86 command would be:

```
-RUN LINK86 MYMOD1.OBJ, MYMOD2.OBJ, P86RN0.LIB, &<cr>
**P86RN1.LIB, P86RN2.LIB, P86RN3.LIB, CEL87.LIB, &<cr>
**E8087, E8087.LIB, LARGE.LIB TO MYPROG.86 BIND <cr>
```

The inclusion of CEL87.LIB, E8087, and E8087.LIB provides support for real arithmetic using the software emulator. This link sequence fully supports all of the features of Pascal-86.

L.2.2 Sample Locate Operations

The following is a sample locate operation using the default settings for controls:

```
-RUN LOC86 SAMPL1.LNK<cr>
```

This sample locate operation binds the logical segments of SAMPL1.LNK to addresses beginning at 00200H (H is for hexadecimal), the default. The output module is called SAMPL1 (the root name of the input module without the LNK extension). Unless you specify a TO clause, the output module (the absolutely located program) will always have the same root name as the input module.

The following is a sample locate operation using the ORDER and ADDRESSES controls:

```
-RUN LOC86 SAMPL2.LNK &<cr>
**ORDER(CLASSES(CODE,STACK,DATA)) &<cr>
**ADDRESSES(CLASSES(CODE(20000H),STACK(4F000H)))<cr>
```

In the invocation line, you can use the ampersand character (&) to continue a long line without executing it.

This sample locate operation collected together the logical segments by class names in the order specified in the ORDER control. The locater then assigned addresses as specified in the ADDRESSES control to the logical segments collected into the CODE and STACK classes. The DATA class received its address assignment from the default algorithm.

L.2.3 Executing Programs

The output module from the locater can be loaded and executed in the 8086 environment by using the Series III RUN command. Position-independent (PIC) and loadtime locatable (LTL) modules produced by LINK86 with the BIND option can also be loaded and executed by the RUN command. These modules could also be used as input to the DEBUG-86 debugger or a similar debugging tool.

To run correctly, a program must be complete, i.e., it must contain all modules necessary to run. For example, in order to run in the Series III 8086 environment with run-time support, a program must contain modules from the run-time support libraries described in 12.2.2. To run in a foreign environment, you must supply your own run-time support and follow the guidelines in Appendix K.

To run a complete program in the Series III 8086 environment, simply use the RUN command. In the example below, both the RUN program and the SAMPL1 program are on drive 0. To refer to any program on a different drive, specify the drive number *d* in the format :F*d*:.

```
-RUN SAMPL1.<cr>
```

Note that in the example, SAMPL1 appears with a period at the end. This period tells the RUN command not to look for an .86 extension. If the program were named "SAMPL1.86", you would not put a period at the end:

```
-RUN SAMPL1<cr>
```

If your program's name has an extension other than .86, you must specify the extension with the name. If its name has an .86 extension, you need not specify it. If its name has no extension, you must specify the final period.

NOTE

If you use the BIND option with LINK86 on a module that is ready to be processed by the RUN loader, and you do not specify its name in a TO clause, the linker will use the root name (and device) of the first file specified as input, but will not append the LNK extension.

L.3 Series III-Specific Compiler Controls

This appendix includes a fold-out page for system-specific examples of most of the Pascal-86 compiler controls. This page is designed to be opened out and used in conjunction with the corresponding text in 10.3.

L.4 Interrupt Handling on the Series III

The Intellec Series III maps the seven Multibus interrupt lines (INT0 through INT7) onto interrupt vector entries numbered 56 through 63; therefore, your application may not use these for software interrupts. Interrupt vector entries available for user software include 64 through 183. Refer to the *Intellec Series III Microcomputer Development System Programmer's Reference Manual*, Order Number 121618 for details.

8087 Support

You may incorporate an 8087 Numeric Data Processor in your Series III by installing the iSBC 337 Multimodule Numeric Data Processor. Refer to the *iSBC 337 Multimodule Numeric Data Processor Hardware Reference Manual* for more information.

NOTE

The Series III Operating System is designed for use by a single operator and supports neither reentrancy nor multitasking.

L.5 Related Publications

Below is a list of other Intel publications you are likely to need in order to use Pascal-86. Most of them describe related Intel products. The manual order number for each publication is given immediately following the title.

- Pascal-86 Pocket Reference, 121541

  A companion to this manual, providing summary information for quick reference.

- *A Guide to the Intellec Series III Microcomputer Development System*, 121632

  A guide to the use of the Series III and associated tools as a total development solution for your iAPX 86 and iAPX 88 microcomputer applications. This tutorial manual takes you through hands-on sessions with the Series III operating system, the CREDIT text editor, the Pascal-86 compiler, the iAPX 86, 88 Family Utilities, the DEBUG-86 applications debugger, and the ICE-86A In-Circuit Emulator.

- *Intellec Series III Microcomputer Development System Product Overview,* 121575

  A summary description of the set of manuals that describe the Intellec Series III development system and its supporting hardware and software. This brief manual includes a description of each manual related to the Series III, plus a glossary of terms used in the manuals.

- *Intellec Series III Microcomputer Development System Console Operating Instructions,* 121609
- *Intellec Series III Microcomputer Development System Pocket Reference,* 121610

  Instructions for using the console features of the Series III, including the DEBUG-86 applications debugger. The *Console Operating Instructions* provides complete instructions, and the *Pocket Reference* gives a summary of this information.

- *Intellec Series III Microcomputer Development System Programmer's Reference Manual,* 121618

  Instructions for calling system routines from user programs for both microprocessor environments, MCS-80/85 and iAPX 86, in the Series III.

- *ISIS-II CREDIT CRT-Based Text Editor User's Guide,* 9800902
- *CREDIT CRT-Based Text Editor Pocket Reference,* 9800903

  Instructions for using CREDIT, the CRT-based text editor supplied with the Series III. The *User's Guide* provides complete operating instructions, and the *Pocket Reference* summarizes this information for quick reference.

- *iAPX 86,88 Family Utilities User's Guide,* 121616
- *iAPX 86,88 Family Utilities Pocket Reference,* 121669

  Instructions for using the 8086-based utility programs LINK86, LIB86, LOC86, CREF86, and OH86 in 8086-based development environments to prepare compiled or assembled programs for execution. The *User's Guide* provides complete operating instructions, and the *Pocket Reference* summarizes this information for quick reference.

- *ASM86 Language Reference Manual,* 121703
- *ASM86 Macro Assembler Operating Instructions,* 121628
- *ASM86 Macro Assembler Pocket Reference,* 121674

  Instructions for using ASM86 in 8086-based development environments. The *Language Reference Manual* gives a complete description of the assembly language; the *Operating Instructions* gives complete instructions for operating the assembler; and the *Pocket Reference* provides summary information for quick reference. You need these publications if you are coding some of your routines in assembly language.

- *PL/M-86 User's Guide*, 121636
- *PL/M-86 Pocket Reference*, 121662
- *Fortran-86 User's Guide*, 121570
- *Fortran-86 Pocket Reference*, 121571

  Instructions for using the PL/M-86 and Fortran-86 languages and compilers in iAPX 86-based development environments. The *User's Guide* gives a complete description of the language and compiler (or translator), and the *Pocket Reference* provides summary information for quick reference. You need these publications if you are coding some of your programs in PL/M-86 or Fortran-86.

- *PSCOPE High-Level Program Debugger User's Guide*, 121790

  Instructions for using PSCOPE, the symbolic debugger for high-level language programs. The *User's Guide* provides complete operating instructions.

- *ICE-86A In-Circuit Emulator Operating Instructions for ISIS-II Users,* 9800714
- *ICE-86A Pocket Reference,* 9800838
- *ICE-88 In-Circuit Emulator Operating Instructions for ISIS-II Users,* 9800949
- *ICE-88 Pocket Reference,* 9800950

  Instructions for using the ICE-86A and ICE-88 In-Circuit Emulators for hardware and software development. The *Operating Instructions* manuals give complete user descriptions of the In-Circuit Emulators, and the *Pocket Reference* guides provide summary information for quick reference. You need the corresponding publications if you are using the ICE-86A or ICE-88 emulator.

- *The iAPX 86,88 User's Manual,* 210201-001

  This manual contains general reference information, application notes, and data sheets describing the 8086, 8087, 8088, and 8089 microprocessors and their use. Extensive discussions of hardware and development software (including PL/M-86, assembly language, LINK86, and LOC86), plus numerous examples of system designs and programs, are included.

- *8087 Support Library Reference Manual*, 121725

  This manual contains specific information on the 8087 support libraries that are available. It includes full descriptions of the DCON87.LIB, CEL87.LIB, and EH87.LIB, as well as a discussion of the IEEE math standard.

- *Run-Time Support Manual for iAPX 86,88 Applications*, 121776

  This manual describes in detail the run-time interface needed to run programs on the iAPX 86,88 family of microprocessors. It includes a description of the run-time libraries required by high-level language compilers, the concepts behind Intel's various operating system environments, the specifications for Intel's Universal Development Interface (UDI), and the definition of the Logical Record Interface (LRI).

Comments	Control	Examples
Causes subsequent code to implement checking in PROG1.SRC	CHECK/NOCHECK	`RUN PAS286 PROG1.SRC CHECK <cr>`
Lists the approximate assembly code for the object code and appends the listing to the source listing in drive 1	CODE/NOCODE	`RUN PAS286 :F1:SOURCE.SRC CODE <cr>`
Generates debug records in the object module PROG.OBJ compiled from the source file PROG.SRC	DEBUG/NODEBUG	`RUN PAS286 PROG.SRC DB <cr>`
Sends error messages to the file specified in the pathname argument SAMPLE.ERR on drive 1	ERRORPRINT/NOERRORPRINT	`RUN PAS286 SAMPLE.SRC EP(:F1:SAMPLE.ERR) <cr>`
Causes extension warnings to occur for any Intel extension to standard Pascal in the file PROG.SRC	EXTENSIONS/NOEXTENSIONS	`RUN PAS286 PROG.SRC NOET <cr>`
Specifies procedures INTERRUPTA and INTERRUPTB as interrupt procedures, but does not generate the interrupt vector for the entry points	INTERRUPT	`RUN PAS286 PROG1.SRC IT(INTERRUPTA INTERRUPTB)<cr>`
Prevents the listing of source lines in the object file SOURCE.OBJ until a LIST control is encountered	LIST/NOLIST	`RUN PAS286 :F1:SOURCE.SRC NOLIST <cr>`
Prevents the creation of an object module from PROG3.SRC	OBJECT/NOOBJECT	`RUN PAS286 PROG3.SRC NOOJ<cr>`
Sends printed output to the file LIST2 on drive 1	PRINT/NOPRINT	`RUN PAS286 :F1:SOURCE.SRC PRINT(:F1:LIST2)<cr>`
Causes the subtitle ‘MODULE ONE’ to appear on every page until another subtitle control (if any) appears in PROG1.SRC	SUBTITLE	`RUN PAS286 PROG1.SRC SUBTITLE(‘MODULE ONE’)<cr>`
Causes the title ‘EUCLIDS GCD PROGRAM’ to appear on every page	TITLE	`RUN PAS286 EUCLID.SRC TT(‘EUCLIDS GCD PROGRAM’) <cr>`
Directs the compiler not to include type records in the file TEMP.OBJ on drive 1	TYPE/NOTYPE	`RUN PAS286 :F1:TEMP.SRC NOTY <cr>`
Produces a cross-reference listing of all identifiers in the file SOURCE.SRC and appends the listing to the default listing file SOURCE.LST on drive 1	XREF/NOXREF	`RUN PAS286 :F1:SOURCE.SRC XREF <cr>`

APPENDIX M
COMPILER INVOCATION AND ADDITIONAL INFORMATION FOR iRMX™ 86 USERS

This appendix contains information that is specific to the iRMX 86 Operating System. It covers the following areas:

- System requirements
- Compiler invocation and file usage
- Examples of compiler control invocation lines
- Examples of system-dependent floating-point library linkage
- Calling iRMX 86 primitives
- Related publications

M.1 Compiler Operation

The Pascal-86 *compiler* is a program that translates your Pascal instructions into object code modules that can be linked and located for execution.

You create a Pascal program by typing instructions into a file using a text editor (such as EDIT) and submitting the file to the Pascal-86 compiler. The file you submit is called a *source file* and the file containing the compiled program is called an *object file*. (The content of the object file is also known as *object code*.) In Pascal-86 you can compile *parts* of a program; each separate compilation is known as an *object module*.

The following discussions assume that you have a suitable copy of the Pascal-86 compiler and an iRMX 86 application system on which the compiler can run. To run the compiler on an iRMX 86 based system, you must have the following hardware and software:

- The iRMX 86 Universal Development System Interface (and other iRMX 86 layers necessary to support the Universal Development System Interface).
- At least one mass storage device. The installation of the compiler always requires a single or double density diskette drive, since the product is delivered in diskette form.
- Enough memory to run the compiler (above and beyond that required for the Operating System). The memory requirement varies for different releases of the compiler, but the code, static data, and dynamic memory requirment is around 150K.

Chapter 1 of this manual leads you through a complete program development sequence using a sample Pascal program. Details of the Operating System environment are provided in the *iRMX 86 Release 5 Operator's Manual.*

M.1.1 Invoking the Compiler

The compiler is supplied on an iRMX 86 format diskette. It may be desirable to copy the compiler to another disk or to one of the directories that the Operating System automatically searches when commands are entered. The compiler consists of a single file: PASC86.

You can invoke the Pascal-86 Compiler from the system console using the standard command format described in the *iRMX 86 Release 5 Operator's Manual.* You can specify continuation lines by using the ampersand (&) as a continuation character. The ampersand can appear any place there is a space or other delimiter.

The invocation command has the general form:

```
-[directory]PASC86 sourcepath[controls]<cr>
```

where

directory is that portion of the pathname that identifies the device and directories which contain the file PASC86. If you omit the directory portion of the pathname, the Operating System automatically searches several directories for the file PASC86. The directories searched and the order of the search are iRMX 86 configuration parameters.

sourcepath is the pathname of the file containing Pascal-86 source code. Refer to the *iRMX 86 Release 5 Operator's Manual* for more information about pathnames.

controls is the optional primary or general compiler controls described in Chapter 10. You can specify many controls in an invocation line if you separate them with spaces (the ampersand (&) creates a continuation line and also separates controls).

<cr> indicates a carriage return.

The following are some examples:

```
-:FD0:PASC86 :WD0:DIR1/MYPROG.SRC PRINT(:LP:) TITLE('TEST 24')<cr>
```

In this example, the compiler resides on device :FD0:. The compiler is directed to compile the source module :WD0:DIR1/MYPROG.SRC, send the output listing to :LP:, and place "TEST 24" in the header on each page of the listing.

```
-PASC86 :F1:PROG1.SOURCE NOPRINT<cr>
```

In this example, the compiler resides in a directory that the Operating System automatically searches; thus, it does not require a device or directory name. The compiler is directed to compile the source module :F1:PROG1.SOURCE. This file resides on the device with the logical name :F1: and has the name PROG.SOURCE. NOPRINT is a compiler control that suppresses all printed output (except error messages) usually generated by the compiler.

M.1.2 Files Used by the Compiler

Input Files

You supply the Pascal source program name for *sourcepath* in the invocation line (see the previous section). You can also include other source files by using the INCLUDE control, as described in section 10.3 of this manual. These files must be standard iRMX 86 named files containing the text of Pascal instructions.

Output Files

By default, the compiler produces two output files (unless you use specific controls to suppress or redirect them): the *listing* file and the *object* file. Also by default, error messages appear at your console and in the listing file.

The listing file (called the PRINT file) contains a listing of the source program, plus any other printed output generated by the compiler as specified by the listing selection controls described in Chapter 10. The object file (sometimes called the object code file or object module) contains the actual code in object module format, which can eventually be executed (after you use the linking and locatiing facilities in Chapter 12). These files are described in more detail in Chapter 11.

The listing file and the object file have the same name as the source file, except that the last pathname component of the listing file has the extension LST and the last pathname component of the object file has the extension OBJ. The compiler creates the files if they do not exist and overwrites them if they do exist. You can optionally change the pathnames for the listing and object files by using the PRINT and OBJECT controls, respectively (described in section 10.3).

For example, if you invoke the compiler using the command line:

```
-PASC86 :PROG:MYPROG.SRC<cr>
```

the compiler creates (or overwrites) the file MYPROG.LST in directory :PROG: to contain the listing, and it creates (or overwrites) the file MYPROG.OBJ in directory :PROG: to contain the object modules.

You can optionally direct certain sections of printed output to files other than the default listing file described previously. In addition to using the PRINT control to specify another file as the listing file, you can specify a different file to receive error messages by using the ERRORPRINT control. Section 10.3 gives details on the use of these controls.

Work Files

The compiler creates and uses work files during its operation which are deleted at the completion of the compilation. All of these files are created in the directory with logical name :WORK:. You should not place your own files in this directory if their names conflict with the compiler's work files. You can set the location of the :WORK: directory during system configuration.

M.1.3 Compiler Messages

When you invoke the compiler, it displays the sign-on message:

```
iRMX 86 Pascal-86, Vx.y
Copyright 1981, 1983, 1984 Intel Corporation
```

where

x	is the version number of the compiler and
y	is the change number within the version.

As the compilation proceeds through its phases, the compiler displays messages that trace the compilation. As each phase starts, the name of the phase is displayed; when the phase terminates, the numeric parameter (if any) and commas are added. The name of the phase is prefixed by NO if the phase was not executed. The name of each phase (except for PARSE and ANALYZE) is the same as the control name that defines the phase.

These messages take the form:

```
PARSE(n), ANALYZE(n), [NO]XREF, [NO]OBJECT
```

where

n is the numer of errors detected during execution of that particular phase.

The output files controlled by the PRINT and ERRORPRINT controls may be directed to the console (:CO:), in which case the compilation trace messages are interrupted with END clauses to show when a phase ends.

When a compilation is finished, the compiler terminates with the message:

```
Compilation of module verdict, n Errors[s] Detected.
End of Pascal-86 Compilation.
```

where

module is the name of the source module,

verdict is either ABORTED or COMPLETED, and

n is the total number of errors detected during compilation.

M.2 Linking, Locating, and Executing on the iRMX™ 86 System

This section shows some examples of linking, locating, and executing Pascal-86 programs in an iRMX 86 environment. When reading this section, keep in mind the following information:

- In order to run a Pascal-86 program in an iRMX 86 environment, you must link it to the iRMX 86 universal development interface (UDI) library called LARGE.LIB.
- The 8087 emulator is not supported for use in an iRMX 86 environment. If you plan to use real arithmetic, you must include the 8087 Numeric Processor Extension in your system and you must link your programs to the libraries 8087.LIB and EH87.LIB.

M.2.1 Sample Link Operations

The following link operation takes two object modules, MYMOD1.OBJ and MYMOD2.OBJ, links them together, and then links in the Pascal run-time libraries to form the output modules MYPROG. To extend the LINK86 command to the next line without transmitting the command, type the ampersand (&) character before the carriage return, and continue typing the command on the next line (do not type the ampersand character between characters of a pathname). The continued line will start with two asterisks (**):

```
-LINK86 MYMOD1.OBJ, MYMOD2.OBJ, P86RN0.LIB, &<cr>
**P86RN1.LIB, P86RN2.LIB, P86RN3.LIB, 87NULL.LIB, &<cr>
**LARGE.LIB TO MYPROG BIND MEMPOOL(+4000H) &<cr>
**SEGSIZE(STACK(+400H))
```

The linker first reads MYMOD1.OBJ and MYMOD2.OBJ for external references and resolves those references between them. Then, the linker attempts to resolve any more external references in the modules by looking at the public symbols in the libraries P86RNO.LIB, P86RN1.LIB, P86RN2.LIB, P86RN3.LIB, 87NULL.LIB and LARGE.LIB. Use of 87NULL.LIB implies that the modules do not perform real arithmetic. The final output module is MYPROG, which can then be loaded and executed on an iRMX 86 system.

If the modules MYMOD1.OBJ and MYMOD2.OBJ perform real arithmetic, you must link them with the 8087 Numeric Processor Extension library (8087.LIB), the library that implements IEEE standarized math features (EH87.LIB), and the library that implements the real built-in functions (CEL867.LIB). This LINK86 command is:

```
-LINK86 MYMOD1.OBJ, MYMOD2.OBJ, P086RN0.LIB, &<cr>
**P86RN1.LIB, P86RN2.LIB, P86RN3.LIB, CEL87.LIB, &<cr>
**8087.LIB, LARGE.LIB TO MYPROG BIND MEMPOOL(+4000H) &<cr>
**SEGSIZE(STACK(+400H))
```

M.2.2 Sample Locate Operations

The following is a sample locate operation using the ORDER and ADDRESSES controls:

```
-LOC86 SAMPLE2.LNK &<cr>
**ORDER(CLASSES(CODE, STACK, DATA)) &<cr>
**ADDRESSES(CLASSES(CODE(2000H), STACK(4F00H)))<cr>
```

In the invocation link, you can use the ampersand character (&) to continue a long line without executing it.

This sample locate operation collects together the logical segments by class names in the order specified in the ORDER control. The locator then assigns addresses as specified in the ADDRESSES control to the logical segments collected into the CODE and STACK classes. The DATA class receives its address assignment from the default algorithm.

If you are locating your program for eventual execution in an iRMX 86 environment, ensure that the addresses you assign to the program do not conflict with the memory reserved for the operating system and your application tasks. Also, ensure that you do not assign your program to memory locations that the Operating System normally assigns as dynamic memory. If you assign absolute addresses to your program, you must reserve those memory locations during iRMX 86 configuration.

M.2.3 Executing Programs

If you have used LINK86 BIND control to produce position-independent code (PIC) and load-time-locatable (LTL) modules, you can load and execute these modules in an iRMX 86 environment by entering their pathnames at the system console. Output from LOC86 can also be loaded and executed in the same manner, as long as you adhere to the restrictions outlined in the previous section.

To run correctly, a program must be complete; that is, it must contain all modules necessary to run. For example, in order to run in an iRMx 86 environment with run-time support, a program must contain modules from the run-time support libraries described in section 12.2.2. To run in a foreign environment, you must supply your own run-time support and follow the guidelines in Appendix K.

To run a complete program in an iRMX 86 environment, simply enter the pathname of the program. In the example that follows, the program SAMPL1 resides in one of the directories that the Operating System automatically searches. The directories searched and the order of search are iRMX 86 configuration parameters. To refer to a program in a different directory, specify the complete pathname of the program.

```
-SAMPL1<cr>
```

M.3 Interrupt Handling in an iRMX™ 86 Environment

Section 8.9 describes several procedures that aid in interrupt processing. Because the iRMX 86 Operating System implements its own form of interrupt processing (refer to the *iRMX 86 Nucleus Reference Manual*), Pascal-86 programs that run in an iRMX 86 environment must not use these Pascal-86 interrupt control procedures.

M.4 Calling iRMX™ 86 Primitives from a Pascal Program

The operating system primitives provided by the various layers of iRMX 86 are available to the Pascal-86 programmer. In order to call an iRMX 86 primitive from a Pascal-86 program, you must follow the instructions for calling a PL/M-86 program from Pascal as described in Appendix J of this manual.

Calling iRMX 86 primitives also requires you to link to the interface libraries associated with the operating system primitives which your application program calls. The names of these interface libraries and order of linkage are described in the *iRMX 86 Configuration Guide.*

M.5 Related Publications

This section lists other Intel publications you might need in addition to this one. The manual order number for each publication immediately follows the title. The paragraph following each title describes the manual.

- *Pascal-86 Pocket Reference*, 121541

  A companion to this manual, providing summary information for quick reference.

- *Introduction to the iRMX 86 Operating System*, 9803124

  A general introduction to the iRMX 86 Operating System. This manual discusses the features of the Operating System and introduces some of the terminology. It also lists and describes each of the manuals in the iRMX 86 manual set.

- *iRMX 86 Release 5 Operator's Manaul*, 172764
- *iRMX 86 Disk Verification Utility Reference Manual*, 144133

  Instructions for entering commands at an iRMX 86 terminal. The operator's manual describes file-naming conventions and provides a complete description of all commands available with the Operating System. The disk verification utility manual describes an interactive utility which examines and restores iRMX 86 volumes.

- *iRMX 86 Human Interface Reference Manual*, 9803202
- *iRMX 86 Nucleus Reference Manual*, 9803122
- *iRMX 86 Release 5 Basic I/O System Reference Manual*, 172766

- *iRMX 86 Release 5 Extended I/O System Reference Manual*, 172767
- *iRMX 86 Loader Reference Manual*, 143318

  Instructions for invoking system calls from user programs.

- *iRMX 86 Programming Techniques*, 142982
- *Guide to Writing Device Drivers for the iRMX 86 and iRMX 88 I/O Systems*, 142926

  Additional information about the Operating System.

- *EDIT Reference Manual*, 143587

  Instructions for using EDIT, the text editor available for use on the iRMX 86 Operating System.

- *iAPX 86,88 Family Utilities User's Guide*, 121616
- *iAPX 86,88 Family Utilities Pocket Reference*, 121669

  Instructions for using the utility programs LINK86, LIB86, LOC86, CREF86, and OH86 in iAPX 86-based environments to prepare compiled or assembled programs for execution. The user's guide provides complete operating instructions and the pocket reference summarizes the information for quick reference.

- *AMS86 Language Reference Manual*, 121703
- *ASM86 Macro Assembler Operating Instructions for 8086-Based Systems*, 121628
- *ASM86 Macro Assembler Pocket Reference*, 121674

  Instructions for using ASM86 in iAPX 86-based development environments. The language reference manual gives a complete description of the assembly language. The operating instructions manual gives complete instructions for operating the assembler. The pocket reference provides summary information for quick reference. You need these publications if you are coding some of your routines in assembly language.

- *PL/M-86 User's Guide*, 121636
- *PL/M-86 Pocket Reference*, 121622
- *FORTRAN-86 User's Guide*, 121570
- *FORTRAN-86 Pocket Reference*, 121571

  Instructions for using the PL/M-86 and FORTRAN-86 languages and compilers in iAPX 86-based development environments. The user's guide gives a complete description of the language and compiler, and the pocket reference provides summary information for quick reference. You need these publications if you are coding some of your programs in PL/M-86 or FORTRAN-86

- *The iAPX 86,88 User's Manual*, 210201

  This manual contains general reference information, application notes, and data sheets describing the 8086, 8087, 8088, and 8089 microprocessors and their use. Extensive discussions of hardware and development software (including PL/M-86, assembly language, LINK86, and LOC86), plus numerous examples of system designs and programs, are included.

- *8087 Support Library Reference Manual*, 121725

  This manual contains specific information on the 8087 support libraries that are available. It includes full descriptions of the DCON87.LIB, CEL87.LIB, and EH87.LIB, as well as a discussion of the IEEE math standard.

- *Run-Time Support Manual for iAPX 86,88 Applications*, 121776

  This manual describes in detail the run-time interface needed to run programs on the iAPX 86,88 family of microprocessors. It includes a description of the run-time libraries required by high-level language compilers, the concepts behind Intel's various operating system environments, the specifications for Intel's Universal Development Interface (UDI), and the definition of the Logical Record Interface (LRI).

GLOSSARY

This glossary defines terms used in the text of this manual. Some of these terms pertain to the Pascal language, some to the compiler, and some to the processor or system environment in which you develop or run programs. For further information concerning a term defined here, refer to the section(s) of this manual that are cited in parentheses after the definition.

argument: a variable, expression, procedure, or function, specified in a procedure or function invocation, that communicates information to that procedure or function. It matches one of the parameters of the invoked procedure or function. (4.1.2, 6.1)

argument list: a list of arguments given in a function designator or a procedure statement. (4.1.2, 7.1.3, 7.2.2)

array: a variable of an array type. (5.3.2)

array type: a structured type consisting of a fixed number of components, or elements, that are all of the same type. (5.3.2)

assignment-compatible: said of an expression and a type if the value of that expression can be assigned to a variable of that type. (5.3.4)

automatic: said of variables that are created and destroyed in accordance with the structure of a program. In Pascal, these are all variables that are declared explicitly within a procedure or function. (5.1)

base address: in an iAPX 86 or iAPX 88 microcomputer system, the 16-bit portion of an address that determines the location of the segment being addressed. (12.3.2)

base type: the type of the components, or elements, of a set type; this must be an ordinal type. (5.3.2)

block: 1. when used in discussing the Pascal language, a section of a source program that includes declarations, definitions, and statements; performs a particular function; and may have other blocks nested within it (or be nested within another block). (2.1) 2. when used in the source listing output from the Pascal-86 compiler, a BEGIN...END, REPEAT...UNTIL, or CASE...OF...END statement group, which may have other such statement groups nested within it. (11.1.2)

block-structured language: a programming language in which the programs are composed of blocks. (2.1)

buffer variable: the implicitly declared variable associated with a file variable and used to hold its current file component. (5.3.2, 5.4.2)

call by reference: communication of an argument to a subprogram (procedure or function in Pascal) by passing the address of the argument. In Pascal-86, this method is used with arguments to variable parameters. (J.1)

call by value: communication of an argument to a subprogram (procedure or function in Pascal) by passing the actual value of the argument. In Pascal-86, this method is used with arguments to value parameters. (J.1)

comment: a sequence of characters, enclosed between a left and a right comment bracketing symbol and including those symbols, which supplies documentation information not to be translated by the compiler. (3.2.1)

compatible: said of two types that may be combined in expressions, or of two parameter lists whose parameter types match. (5.3.4, 6.4.7)

compiler: the program that translates your Pascal instructions into object code ready to be linked and located. (11)

component type: the type of the components, or elements, of an array type. (5.3.2)

conditional compilation: process whereby the compiler skips selected portions of the source file if specified conditions are not met.

conditional statement: a statement that selects for execution one of its component statements; in Pascal, an IF or CASE statement. (7.2.4)

constant: a data item that cannot change during program execution. It may be a literal constant (e.g., a signed integer or a literal string) or a named constant referred to by an identifier. (5.1, 5.2)

control: an instruction to the compiler, given either in the compiler invocation line or in a control line within the source text. (10)

control line: a line, within the text of a source program, that starts with a dollar sign in the leftmost column, and contains compiler controls rather than constructs that are part of the Pascal program. (10.1)

current file component: the one component of a file variable that is accessible at a given time. (5.3.2, 5.4.2)

declaration: a program construct that introduces an object having meaning at run time, such as a variable, procedure, function, or label. (2.1, 5)

default value: the input parameter value or control value that is assumed by a program (such as a compiler) if no value is explicitly given. (10.1)

definition: a program construct that introduces an object having meaning at compile time, such as a constant or a type. (2.1, 5)

denormalized: said of a real value if it has a zero exponent and a zero explicit or implicit leading bit, but is not a normal zero. (14.7)

dereferencing a pointer: accessing a dynamic variable by referring to its associated pointer variable. (5.3.3)

directive: a word symbol that has a special meaning, but which can also be defined as an identifier in programs. In Pascal-86, only one directive (FORWARD) is defined. (6.5, F)

driver: a routine that transfers data or performs other communication with an external device. (K.2.1)

dynamic: said of variables that are generated at run time by statements within a program, rather than being declared explicitly. (5.1, 5.3.3)

dynamic symbol table: the compiler's dictionary. Information about all the symbols is recorded here. It is dynamic because it can grow from main memory to mass storage. (10.3.20)

empty set: the set (object of a set type) that contains no elements. (5.3.2)

empty statement: a statement that contains no symbols and performs no action. (7.2)

entire variable: a complete variable of a given type, rather than a component of a structured variable. (5.4.2)

enumerated type: a simple, ordinal type that defines an ordered set of values by listing the identifiers that denote these values. (5.3.1)

error: 1. in the Pascal-86 compiler output listing, a mistake in the source program that is severe enough to prevent generation of an object module. (11.1.3, 13.1) 2. a run-time condition that may cause the output of a program to be wrong, due to a logical mistake in the source program or due to invalid input. (14.1)

exception: a run-time condition that may cause the output of a program to be wrong, due to a logical mistake in the source program or due to invalid input; also called a run-time error. The use of the term "exception" implies that in some cases, a routine can be called to handle the situation, and then processing can continue normally. (14.1)

executable: said of the parts of a program that cause the processor to perform actions at run-time. In Pascal, statements are executable, whereas definitions and declarations are not. (7.2)

exponent: in the internal representation of a real value, the part that designates (in binary) the base-2 exponent of the number. (7.1.8, H.1)

expression: a language construct, used in statements, which is evaluated at run-time. (7.1)

extension: a language feature not present in the corresponding standard language; in this manual, a feature in Pascal-86 that is not part of the Draft Proposed Standard Pascal (ISO/DP 7185). (1.2.4, 11.1.3)

external: said of a procedure or function that is referenced in the module being compiled, but is declared in another module. (4.2.3)

fatal error: an anomaly in the compiler or compile-time environment that makes it impossible to proceed with the compilation. (11.1.3, 13.1)

field: a component, or element, of a record type. (5.3.2)

field designator: the construct used to reference a component, or field, of a record variable. (5.3.2, 5.4.2)

field identifier: the identifier by which a field of a record is referenced. (5.3.2)

file: 1. a variable of a file type (5.3.2). 2. a collection of information on a physical input or output device. (5.3.2, J.2)

file descriptor: a run-time data structure containing the attributes of a physical file; one exists for every file that is preconnected or open. (K.2.1)

file type: a structured type consisting of a sequence of components that are variable in number, are all of the same type, and are accessed sequentially. (5.3.2)

floating-point number: see *real number.*

function: a subprogram, or subordinate block, that returns a value and that is invoked by using its function designator in an expression. (2.1, 6)

function designator: a language construct, used as an operand in an expression, that invokes a function. It consists of the function name, followed by a parameter list if needed. (7.1.3)

functional parameter: a function that is used as a parameter of a procedure or another function. (6.4.6)

general control: a control that may appear either in the invocation line or in a control line anywhere in the source program text, and that may be changed later in the source program. (10.1)

global: said of program objects that are declared or defined at the outer level of a program block or a module. (4.1.2, 6.1)

group: in an iAPX 86 or iAPX 88 microcomputer system, a collection of logical segments that must be located within a 64K-byte range; that is, within a memory segment. (12.3.2)

heap: a pool of memory, provided through the Pascal-86 run-time system, from which storage for dynamic variables is taken. (5.3.3)

host type: the ordinal type on which a subrange type is based. (5.3.1)

identifier: a sequence of letters or digits, beginning with a letter, that denotes a module, procedure, function, constant, type, variable, parameter, or field designator. (3.3.1)

INCLUDE file: a separate file of source code that is inserted, or included, in the source input to the compiler by means of an INCLUDE control in the source text. (10.2, 10.3)

index type: the type of the array selector, or index, used to reference an element of an array; it must be an ordinal type. (5.3.2)

indexed variable: the construct used to reference a component of an array variable. (5.3.2, 5.4.2)

integer (literal integer): a sequence of digits, optionally terminated by a numberbase symbol (B, b, Q, q, H, or h), that represents a decimal, binary, octal, or hexadecimal integer. (3.3.2)

integer type: an ordinal type that represents a subrange of the whole numbers. (5.3.1, 5.3.4)

interface specification: the part of a module that follows the module heading and defines the interface between that module and other modules. This interface consists of the objects in other modules that may be referenced by this module, and the objects in this module that may be referenced by other modules. (4.2.3)

interrupt: a signal, sent when an external event occurs, that causes a microprocessor CPU to stop what it is doing, perform a special routine to handle the interrupt, and then resume the interrupted processing where it left off. (8.9.1, 10.3, K.1)

interrupt procedure: a procedure that is called when an interrupt occurs, to handle that interrupt. (8.9.1, 10.3, K.1)

interrupt vector: an array of absolutely-located entries, starting at memory location 0, that are reserved for the addresses of interrupt procedures, one for each interrupt number. (10.3, K.1)

invocation line: the line of text used to invoke the compiler. (system-dependent appendix)

keyword: a word symbol that has a special meaning and cannot be defined as an identifier in programs. (3.1, F)

label: a sequence of decimal digits (decimal integer) used to mark a statement so that a GOTO statement may refer to it. (3.3.4)

lazy input: an input buffering scheme whereby the run-time system fills the buffer variable for an input file only after it ensures that new data has been requested. This scheme differs from the standard Pascal buffering method, which involves reading ahead in the file to allow overlapping of input with computations. Lazy input is necessary for interactive programs, to ensure that a program does not query the terminal for input until it has prompted the user for that input. (8.7)

library: a file that contains object modules and is created and maintained by a library utility such as LIB86. (12.2.1)

line marker: the implementation-defined character or character sequence that indicates the end of a line in a text file. In Pascal-86, this is an ASCII carriage return followed by an ASCII line feed. (5.3.2)

link: to combine together several compiled or assembled object modules to prepare them for locating and execution. (12.1, 12.2)

listing file: the file containing printed output produced by the compiler. (system dependent appendix)

literal integer: see *integer.*

literal real number: see *real number.*

literal string: see *string.*

load-time locatable (LTL) code: object code that can be located at load time using the RUN loader, and that may refer to segment base addresses to access other segments in memory. Position-independent code (PIC) is always load-time locatable, but load-time locatable code is not always position-independent. (12.3.3)

local: said of program objects that are declared or defined within a subordinate block in a program, rather than at the outer level. (4.1.2, 6.1)

locate: to bind the code in linked object modules to memory addresses so that it may be executed. (12.1, 12.3)

logical blank: a blank or blank substitute that may separate symbols in a program. (3.2)

logical record: a unit of data containing exactly one component of a non-text file, or one line of a text file. This is the unit of data that is processed by the Pascal-86 run-time system. (K.2)

logical record interface: the software interface between the Pascal-86 run-time support libraries and the operating system. (K.2)

logical segment: a piece of an object module that is acted on by the programs that link and locate the object module. (12.1)

LTL: see *load-time locatable (LTL) code.*

main program module: see *program module.*

memory segment: see *segment.*

module: a separately compiled program unit. In Pascal-86, it may include a module heading and interface specification; global declarations for labels, data, procedures, and functions; and/or a main program block. (2.2, 4, I)

module heading: a program construct that begins a module and identifies it by name. (4.2.2)

named constant: a constant that has been associated with an identifier by defining it in a constant definition. (5.2)

NaN: in real (floating-point) number representation, one of several bit patterns that are not actual numbers. (NaN stands for "Not a Number.") The presence of NaNs in a computation generally indicates a real arithmetic exception condition. (7.1.8)

nested: contained; said of a procedure or function block that lies within an enclosing procedure or function block, of a statement block (BEGIN...END, REPEAT... UNTIL, or CASE...OF...END) that lies within another statement block, and of an INCLUDE control invoked because of a file brought in by a previously invoked INCLUDE control. (2.1, 11.1.2)

neutral object: in a Pascal-86 program being compiled, an object whose type has been "neutralized" by the compiler because of a program error. This object is treated as compatible with any type, so that subsequent references to an incorrect type specification do not cause additional errors to be reported. (13.3)

non-main module: a module other than the program module, that is, a module that does not contain the main program. (4.1.1)

nonterminal symbol: in the syntax notation or other specification of a programming language, a term (such as *identifier* or *expression*) standing for a language element or construct that must be filled in by the programmer. (Preface, page viii)

normal zero: the real value whose exponent is of minimum value and whose significand is all zeros. (14.7)

normalized: said of a real value if it is a normal zero or if its leading significand bit is one and the exponent is greater than zero. (14.7)

object: one of the following user-defined or predefined entities in a Pascal program: module, procedure, function, constant, type, variable, parameter, field designator, or label. (3.3.1)

object code: program code that has been processed by an assembler or compiler, and that may have been processed further by a linking and/or a locating program. (system-dependent appendix)

object file: a file containing compiled object code. (system-dependent appendix)

object module: the compiled code output by one separately compiled Pascal source module. (system-dependent appendix)

offset: in an iAPX 86 or iAPX 88 microcomputer system, the 16-bit portion of the address that gives the position of the addressed location within the segment specified by the base address. (12.3.2)

operand: an item, such as a variable or constant of some type, that can be evaluated and is combined with other such items in an expression by means of operators. (7.1)

operator: a language element specifying an operation to be performed on one or two operands in an expression. (7.1)

ordinal type: a simple type whose values can be mapped onto a subset of the nonnegative integers. (5.3.1)

parameter: a program object, named in a program, procedure, or function heading, that provides communication between the program, procedure, or function and its environment. In the case of a procedure or function, the parameter is matched by an argument in the statement that invokes the procedure or function. (4.1.2, 6.4)

parameter list: a list of parameter identifiers given in the heading of a program, procedure, or function. (4.1.2, 4.2.1, 6.4.1)

physical record: a unit of data occupying one division of a physical I/O device; for instance, a block on a disk. (K.2)

PIC: see *position-independent code.*

pointer type: a type whose variables are used to reference dynamic variables. (5.1, 5.3, 5.3.3)

portable: said of programs that can be transported from one processor to another without modification. (1.2.3)

position-independent (PIC) code: object code that can be located at load time using the RUN loader, and that does not refer to segment base addresses in order to access other segments in memory. Position-independent code is always load-time locatable (LTL), but load-time locatable code is not always position-independent. (12.3.3)

precedence: the rule determining in what order the operators in an expression are to be applied when the expression is evaluated. (7.1)

preconnection: the association of a physical file with a logical file (file variable) before a program is run; that is, at invocation time. (12.4.1)

predefined: said of a program object—such as a program parameter, type, constant, procedure, or function—whose definition or declaration is part of the Pascal-86 language, so it need not be defined or declared in programs. Also said of an identifier that stands for such an object. (3.3.1)

primary control: a control that must appear either in the invocation line or in a control line preceding the first non-control line in the source program text, and that serves as a global control affecting the entire compilation. (10.1) Compare with *general control.*

private: said of a program object that may be referenced only by the module in which it occurs; also said of a symbol (identifier or label) that stands for such an object. (4.2.3)

private heading: in a non-main module, the heading that begins the private section of a module and gives the module a name. (2.5, 4.2.4)

private section: the part of a module in which private objects—objects which only that module may reference—are defined. (4.2.3)

procedure: a subprogram, or subordinate block, that is invoked by a procedure statement. (2.1, 6)

procedural parameter: a procedure that is used as a parameter of a function or another procedure. (6.4.5)

program block: the block that defines the main program, with which execution will start when the program is run. (2.1, 4.1.1)

program heading: in a program module, the heading that begins the program block and gives the program module a name. (2.5, 4.2.1)

program module: the module that contains the main program, with which execution will start when the program is run. (2.2, 4.1.1)

program parameter: a parameter, defined in a program heading, that denotes an input or output file to be used by the program. (4.1.2, 4.2.1)

public: said of a program object that other designated modules may freely reference; also said of a symbol (identifier or label) standing for such an object. (4.2.3)

public section: a part of a module in which public objects are defined. (4.2.3)

real number (literal real number): a sequence of digits and symbols representing a decimal number that includes a fractional part. It specifies a significand (integral part) and either an exponent, a sequence of fraction digits, or both. (3.3.3)

real type: a simple, non-ordinal type that represents a real, or floating-point, number. (5.3.1, 5.3.4)

record: 1. a variable of a record type. (5.3.2) 2. see *logical record.* 3. see *physical record.*

record type: a structured type consisting of a fixed number of components (fields), possibly of different types, that are referenced by means of identifiers. (5.3.2)

record variant: one of several different structures, or sets of fields, that data items of a record type may assume. (5.3.2)

recursive: said of a procedure or function that calls itself, either directly or indirectly. (2.1)

reentrant: said of a procedure or function that can be interrupted, then called again from an interrupt handler or another task before the first invocation is finished. (2.1)

referenced variable: the construct used to reference a dynamic variable by means of its associated pointer variable. (5.3.3, 5.4.2)

repetitive statement: a statement that specifies that its component statements are to be executed repeatedly; in Pascal, a WHILE, REPEAT, or FOR statement. (7.2.6)

scalar type: see *simple type.*

scope: of a definition or declaration, the portion of the source program over which the stated symbol-object association holds. (4.1.2, 6.1)

segment: in an iAPX 86 or iAPX 88 microcomputer system, an area of memory that starts at a 16-byte boundary, consists of up to 64K contiguous bytes, and can be addressed without changing the base register used. (12.3.2)

semantics: the set of rules for determining, given a syntactically acceptable program in a given language, what that program means—that is, what actions it will cause the processor to take. (3)

set: a variable of a set type. (5.3.2)

set constructor: an expression, consisting of a list of *elements* enclosed between square brackets, which represents a set. (7.1.2)

set type: a structured type whose values represent a collection of objects of an ordinal type. (5.3.2)

sign bit: in the internal representation of a real value, the bit that indicates the sign of the number. (7.1.8, H.1)

signed integer: an integer (literal integer) preceded by an optional plus or a minus sign. (3.3.2)

signed real number: a real number (literal real number) preceded by an optional plus or a minus sign. (3.3.3)

significand: in the internal representation of a real value, the part that designates (in binary) the significant bits of the number. (7.1.8, H.1)

simple statement: a statement, such as an assignment statement or a procedure statement, that contains no other statements. (7.2)

simple type: a type whose variables have a single value; also called *scalar type.* (5.3, 5.3.1)

source code: code written in a programming language such as Pascal. (system-dependent appendix)

source file: a file containing source code. (system-dependent appendix)

statement: 1. when used in discussions of the Pascal language, a program construct that defines an action to be performed by the program. (2.1, 2.5, 7.2) 2. when used in the source listing output from the Pascal-86 compiler, a construct delimited by a semicolon not within a parameter list or beginning with DO, THEN, ELSE, OTHERWISE, UNTIL, or the OF in a CASE statement. (11.1.2)

static: said of variables that exist for the entire run of a program. In Pascal-86, only global variables are static. (5.1)

static symbol area: memory used by the compiler for context information necessary because of block structure. (10.3.20)

string: a sequence of one or more characters enclosed by apostrophes, representing a value of type CHAR (if a single character) or of type PACKED ARRAY [1..n] OF CHAR, where *n* is greater than 1 and is equal to the number of enclosed characters. (3.3.5)

string type: a structured type defined as PACKED ARRAY [1..n] OF CHAR, where *n* is a positive integer. (5.3.2, 5.3.4)

strongly typed: said of a programming language, such as Pascal, in which every data item must have a type, and in which strict type compatibility rules must be followed. (5.3)

structured statement: a statement, such as a compound statement or a REPEAT statement, that contains one or more embedded statements. (7.2)

structured type: a type whose variables are made up of a number of single values; such a type is built up from simple types. (5.3, 5.3.2)

subrange type: a simple, ordinal type defined as a subrange of an ordinal type. (5.3.1)

syntax: the set of rules defining what sequences of symbols make up acceptable programs in a given programming language. (3)

tag field: in a record containing record variants, a field, common to all variants, which identifies the variant assumed at a given time. (5.3.2)

terminal symbol: in the syntax notation or other specification of a programming language, a symbol (such as a keyword, letter symbol, or punctuation symbol) that is to be used verbatim in programs. (Preface, page vii)

text file: a file type that has components of type CHAR and is subdivided into lines by means of implementation-defined line markers. (5.3.2)

token: a program symbol, such as a keyword, punctuation symbol, identifier, or literal string, that is divisible from other such symbols by one or more logical blanks. (3.3)

type: a program object denoting a set of values which a data item can assume. Every constant, variable, parameter, and expression in Pascal has a type. (5.1, 5.3)

type constructor: a keyword specifying a structuring method for a structured type. There are four type constructors in Pascal: ARRAY, RECORD, SET, and FILE. (5.3.2)

unnormalized: said of a real value in the temporary real format if its exponent is greater than zero and its explicit leading bit is zero. (14.7)

value parameter: in a procedure or function declaration, a parameter whose argument is evaluated once, when the procedure or function is invoked, and is not changed by the procedure or function. Its argument may be any expression of the proper type. (6.4.2)

variable: a data item whose values can change during program execution, and which the program processes. It is referred to by an identifier. (5.1, 5.4)

variable parameter: in a procedure or function declaration, a parameter whose argument may be changed by the procedure or function. Its argument must be a variable. (6.4.3)

variant: see *record variant.*

variant record: a record type with record variants. (5.3.2)

warning: in the output from the compiler, a message indicating an anomaly in programming style that may point to a problem, but that will not affect the validity of the compiled object code. A warning is less serious than an error. (11.1.3, 13.1)

workfile: a temporary file created by a program (such as a compiler) for its own internal use and deleted when the program finishes. (system-dependent appendix)

INDEX

REQUEST FOR READER'S COMMENTS

Intel's Technical Publications Departments attempt to provide publications that meet the needs of all Intel product users. This form lets you participate directly in the publication process. Your comments will help us correct and improve our publications. Please take a few minutes to respond.

Please restrict your comments to the usability, accuracy, readability, organization, and completeness of this publication. If you have any comments on the product that this publication describes, please contact your Intel representative. If you wish to order publications, contact the Intel Literature Department (see page ii of this manual).

1. Please describe any errors you found in this publication (include page number).

2. Does the publication cover the information you expected or required? Please make suggestions for improvement.

3. Is this the right type of publication for your needs? Is it at the right level? What other types of publications are needed?

4. Did you have any difficulty understanding descriptions or wording? Where?

5. Please rate this publication on a scale of 1 to 5 (5 being the best rating) ______

NAME ______ DATE ______

TITLE ______

COMPANY NAME/DEPARTMENT ______

ADDRESS ______

CITY ______ STATE ______ ZIP CODE ______

(COUNTRY)

Please check here if you require a written reply □

WE'D LIKE YOUR COMMENTS ...

This document is one of a series describing Intel products. Your comments on the back of this form will help us produce better manuals. Each reply will be carefully reviewed by the responsible person. All comments and suggestions become the property of Intel Corporation.

Intel Corporation
Attn: Technical Publications M/S 6-2000
3065 Bowers Avenue
Santa Clara, CA 95051

intel®

INTEL CORPORATION, 3065 Bowers Avenue, Santa Clara, California 95051 (408) **987-8080**

Printed in U.S.A.