[illegible]

PASCAL-80 USER'S GUIDE

Manual Order Number: 9801015-02

Intel will provide limited telephone assistance to the customer in the understanding of the operation of the product.

In addition, if a problem is encountered which the user diagnosis indicates is caused by a defect in this Vendor Supplied Product, the user is requested to fill out a Problem Report form and mail it to:

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

Intel will use its best efforts to respond to Problem Reports in one of the following ways: 1) Release information to correct the problem, 2) Offer a new revision, when available, with corrected code to fix the problem, or 3) Issue a notice of availability of a new revision with corrected code.

A235/780/5K/FL

PREFACE

This manual describes the PASCAL-80 language for programming the 8080 and 8085 microcomputers. PASCAL-80 is based on the Pascal language invented in 1970 by Professor Niklaus Wirth of Zurich, and incorporates extensions designed to take advantage of both the microcomputer features of the 8080 and 8085, and the system features of ISIS-II.

Each release of PASCAL-80 is characterized by a two-digit code in the form of:

"version number" . "release number within version"

Manual Organization

The PASCAL-80 language is a superset of the Pascal programming language, as defined in *PASCAL User Manual and Report*, Second Edition, by Kathleen Jensen and Niklaus Wirth (Springer-Verlag, 1974, Corrected Printing, 1978). This book defines "Standard Pascal" and will be referred to in the remainder of this manual as PASCAL User Manual. Since PASCAL User Manual is a concise reference for the features of Standard Pascal, the programming language PASCAL-80 will be described only by its differences from, and extensions to, Standard Pascal, the assumption being that the reader is familiar with the contents of PASCAL User Manual, and moderately conversant with Standard Pascal.

Chapters 1 through 3 describe the features of the PASCAL-80 language, and Chapter 4 describes how to operate the PASCAL-80 System.

Chapter 5 presents example programs illustrating the features of PASCAL-80. The appendices supply reference information for all aspects of program generation, compilation, and execution.

Audience

This manual is intended for programmers who are familiar with PASCAL. It contains the following chapters and appendices:

"Chapter 1. Introduction," which gives a general description of the PASCAL-80 system.

"Chapter 2. Language Features," which describes the data types, control constructs, and procedure and function declarations.

"Chapter 3. Predeclared Procedures," which describes the predeclared procedures and functions in PASCAL-80.

"Chapter 4. Operating Instructions," which describes how to generate, compile, and execute a PASCAL-80 program.

"Chapter 5. Example Programs," which presents four PASCAL-80 programs, complete with compiler listing and output.

"Chapter 6. Separate Compilation, Program Linkage, Relocation, and Execution," which gives general instructions for partitioning a Pascal program into separate compilation units, for linking to external non-Pascal object modules and for creating Load and Go versions of Pascal programs.

"Chapter 7. Preparing Programs to Run Under RMX/80," which gives information applicable to running Pascal programs in the RMX/80 environment.

"Chapter 8. Run-Time Error Procedures," which gives instructions for declaring and using user-defined error procedures.

"Appendix A. Summary of Operations," which presents, in tabular form, the allowable operations among data types.

"Appendix B. PASCAL-80 Vocabulary," which contains all of the operators and reserved words in the PASCAL-80 programming language.

"Appendix C. Summary of Compiler Directives," which describes the directives for the compiler.

"Appendix D. Compiler Error Messages," which lists all errors that can be signaled during the compilation of a Pascal program.

"Appendix E. Run-Time Errors," which is a listing of the error messages issued by the system default error procedure.

"Appendix F. ISIS-II Error Messages," which is a listing of the error messages issued by ISIS-II.

"Appendix G. Syntax Summary," which summarizes the syntax of legal PASCAL-80 programs.

"Appendix H. Summary of Extensions to Standard Pascal," which summarizes all extensions to Standard Pascal provided by PASCAL-80.

"Appendix I. Implementation Details," which presents the format of the predeclared data types in PASCAL-80, lists certain implementation size limits, and summarizes the differences between PASCAL-80 and Standard Pascal.

"Appendix J. Pascal Reference Texts," which lists a number of books that are of interest to the Pascal programmer.

"Appendix K. Directory of Release Diskettes," which lists the file directories of both the single density and double density release diskettes.

Related Publications

For additional information concerning ISIS-II and PASCAL-80, refer to the following Intel documents:

ISIS-II User's Guide	Order Number: 9800306
8080/8085 Floating Point Arithmetic Library User's Manual	Order Number: 9800452
ISIS-II CREDIT (CRT-Based Text Editor) User's Guide	Order Number: 9800902
RMX/80 User's Guide	Order Number: 9800522
RMX/80 Interactive Configuration Utility User's Guide	Order Number: 142603
PL/M Programming Manual	Order Number: 9800268
ISIS-II PL/M Compiler Operator's Manual	Order Number: 9800300
8080/8085 Assembly Language Reference Manual	Order Number: 9800301

ISIS-II 8080/8085 Macro Assembler Operator's Manual	Order Number: 9800292
FORTRAN-80 Programming Manual	Order Number: 9800481
ISIS-II FORTRAN-80 Compiler Operator's Manual	Order Number: 9800480

Notation in this Manual

Throughout this manual, the following notation is used to describe the format of a Pascal program:

1. Certain key words are reserved and cannot be used as identifiers. In this manual they are in capital letters. Refer to Appendix B for a list of reserved words.
2. Variable names are denoted by words in lower case.
3. Language features which exceed Standard Pascal are shaded.

CONTENTS

CONTENTS (Cont'd.)

TABLES

ILLUSTRATIONS

CHAPTER 1 INTRODUCTION

General Description

Pascal is a highly structured computer programming language devised in 1970 by Professor Niklaus Wirth of Zurich. It is a very powerful high-level language which is now gaining wide acceptance as a useful programming tool.

PASCAL-80 is a Pascal language system designed for the Intel 8080/8085 microcomputers under control of the ISIS-II Operating System. It is designed to run on a 64K byte Intellec or Series II microcomputer development system plus CRT and floppy diskettes.

Although a minimum system would include only one single-density diskette, for maximum efficiency a system with at least two double-density diskettes is recommended.

The PASCAL-80 System is composed of a Run-Time System (RTS) and a compiler which itself executes on the Run-Time System. The PASCAL-80 compiler converts Pascal source code into an intermediate P-Code (pseudo-code), which is then interpreted by the Run-Time System.

The ISIS-II text editor is used to create the Pascal source file(s). The PASCAL-80 compiler is then used to compile the source programs so that they may be executed on the RTS. A listing of the PASCAL-80 source program is provided by the compiler during compilation. Any error messages are included in this listing.

Complete operating instructions can be found in Chapter 4, along with a more complete description of the structure of the PASCAL-80 System. Chapter 5 presents four PASCAL-80 example programs, including a complex example, along with typical user interaction with these programs. All example programs are listed in the manual and are also present on the PASCAL-80 Release Disk in source form.

PASCAL-80 programs may be loaded and executed under control of the PASCAL-80 Command Line Interpreter, as described in Chapter 4, or they may be converted to standard ISIS-II object modules and executed directly in the ISIS-II environment. The procedure for generating these Load and Go versions is described in Chapter 6. Also described in Chapter 6 are the PASCAL-80 extensions which allow a Pascal program to link with non-Pascal object modules coded in PL/M-80, FORTRAN-80, or ASM-80.

Chapter 7 describes the procedures necessary to generate an RMX/80 applications system which includes a Pascal task. The RMX/80 Interactive Configuration Utility, which automates much of this process, is also discussed.

A powerful feature of the PASCAL-80 System is the ability for a Pascal program to declare an error procedure which is invoked upon the detection of a run-time error. Use of this facility, which is described in Chapter 8, is extremely important in applications systems, since it allows a program to retain control over its execution, even in the event of run-time errors.

The PASCAL-80 Language

The PASCAL-80 language is a superset of the programming language Pascal, as defined in *PASCAL User Manual and Report*, Second Edition, by Kathleen Jensen and Niklaus Wirth (Springer-Verlag, 1974, Corrected Printing, 1978). This book defines "Standard Pascal" and will be referred to in the remainder of this manual as PASCAL User Manual. Since PASCAL User Manual is a concise reference for the features of Standard Pascal, the programming language PASCAL-80 will be described only by its differences from, and extensions to, Standard Pascal, the assumption being that the reader is familiar with the contents of PASCAL User Manual, and moderately conversant with Standard Pascal.

System Structure

As mentioned, the PASCAL-80 Run-Time System is an interpreter, which executes P-Code generated by the PASCAL-80 compiler. A detailed description of the structure of the P-Code can be found in Appendix I.

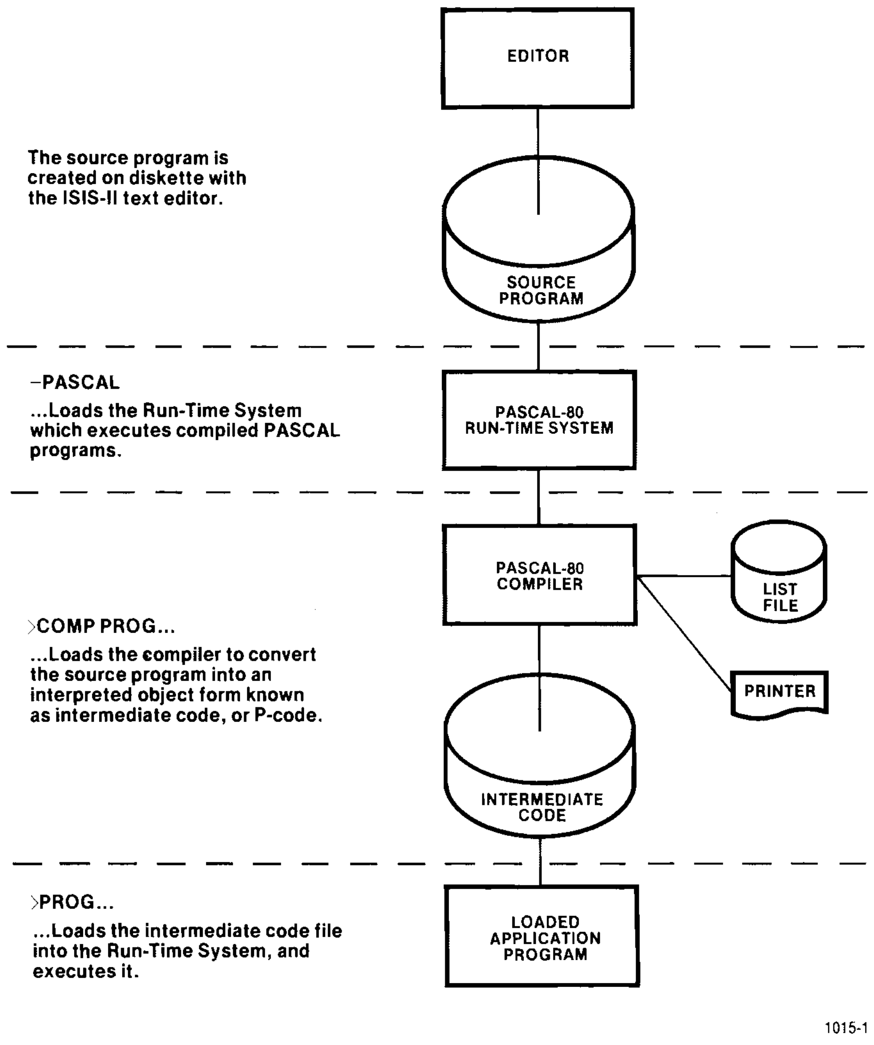
PASCAL-80 programs can be partitioned into SEGMENT (overlay) procedures, and, therefore, very large programs may be executed. This powerful PASCAL-80 extension is described in Chapter 2.

The PASCAL-80 System incorporates a powerful program tracing facility which allows the programmer to selectively monitor the execution of a Pascal program, and to interrupt execution at any point.

PASCAL-80 allows a user error procedure to be specified, so that a program may retain control in the event of a run-time error, and perform error recovery procedures.

Program Development Cycle

The cycle for development and execution of PASCAL-80 application programs that must be performed by the programmer is as shown in figure 1-1.



The source program is created on diskette with the ISIS-II text editor.

−PASCAL
...Loads the Run-Time System which executes compiled PASCAL programs.

>COMP PROG...
...Loads the compiler to convert the source program into an interpreted object form known as intermediate code, or P-code.

>PROG...
...Loads the intermediate code file into the Run-Time System, and executes it.

Figure 1-1. Program Development Cycle

CHAPTER 2 LANGUAGE FEATURES

As mentioned in the Introduction, PASCAL-80 is an extension of Standard Pascal, as defined in the PASCAL User Manual, and it is assumed that the reader is familiar with the structure and characteristics of Standard Pascal. In this chapter we will describe the features of PASCAL-80 in terms of Standard Pascal, and present examples of all language extensions.

Program Structure

The complete, detailed PASCAL-80 syntax is presented in Appendix G; the following briefly describes the overall structure of a Pascal program.

Every PASCAL-80 program consists of a heading, followed by a block, which in turn is followed by the program terminator, a ".". The block contains a number of declaration parts, which define all objects local to the block, followed by the statement part, which specifies the actions to be performed on the declared objects.

```
<program>     ::=  <program heading> <block>
<block>       ::=  <label declaration part>
                   <constant declaration part>
                   <type declaration part>
                   <variable declaration part>
                   <procedure and function declaration part>
                   <statement part>
```

Program Heading

The program heading names the program, which allows the predeclared procedure "exit(<name>)" (described below) to force the program itself to cease execution. Any file identifiers, if present in the heading, are ignored.

```
   <program heading> ::= PROGRAM <identifier>
or <program heading> ::= PROGRAM <identifier>(<file identifier list>)
```

The following are legal program headings:

```
PROGRAM sort;
PROGRAM convert(input,output);
PROGRAM generate(listing);
```

Label Declaration Part

Any statement in a PASCAL-80 program may be labelled with an unsigned integer label followed by a colon. This allows references to this statement by goto statements. Any such label must be declared in the label declaration part, which has the form:

```
LABEL <label> { , <label> }
```

where <label> is an unsigned integer of at most four digits. Examples of label declarations are:

```
LABEL  1,2,3,4,5;
LABEL  1999,2999,6789;
```

Constant Declaration Part

A constant declaration defines an identifier as a synonym for a constant. The general form for this declaration is:

```
CONST <identifier> = <constant>;  { <identifier> = <constant>; }
```

where <constant> is either a number, a constant string, or a constant identifier. Examples of constant declarations are:

```
CONST maxfiles = 6;
CONST version = 'PASCAL-80 V1.0'; nfiles = maxfiles;
```

Type Declaration Part

The type declaration part is a mechanism for creating new data types. The declaration determines a set of values for the new type and associates an identifier with that set. The general form for this declaration is:

```
TYPE <identifier> = <type>;  { <identifier> = <type>; }
```

Examples of type declarations are:

```
TYPE weekdays = (monday,tuesday,wednesday,thursday,friday);
TYPE filenumber = 1. .nfiles; filetype = (source,object);
```

Variable Declaration Part

Every variable used in a program statement must first be declared in the variable declaration part. This declaration associates both an identifier and a data type with the variable being declared. The general form of this declaration is:

```
VAR  <identifier>  { ,<identifier> }  :  <type>;
    { <identifier>  { ,<identifier> }  :  <type>; }
```

Examples of variable declarations are:

```
VAR alpha,beta,gamma:  integer;
VAR name: string[30];
```

Procedure and Function Declaration Part

Every procedure and function must be defined before its use. A procedure declaration associates an identifier with a set of actions, which may then be invoked by referring to the identifier. Functions are similar to procedures but also yield a result value, and therefore can be referred to within expressions. Procedure and functions are discussed below in more detail.

Data Types

Every variable appearing in a Pascal program must be associated with one and only one data type. The following data types are predeclared in the PASCAL-80 System; using the type declaration facility, new data types of arbitrary complexity may be defined.

In the following discussion, a word is defined to be a two-byte quantity, aligned on an even byte boundary.

The Boolean Type

A variable of type boolean can have one of the values denoted by the predefined identifiers false and true. The definition of the type is:

```
TYPE boolean = (false,true);
```

It should be noted that under this definition, false<true.

A variable of type boolean requires two bytes of storage, although only the least significant bit is significant.

The Integer Type

The standard type integer is defined to be a subrange of whole numbers. In PASCAL-80 its definition is:

```
TYPE integer = -32768..32767;
```

There also exists a predefined standard variable of type integer named **maxint** which has the value 32767.

A variable of type integer occupies two bytes of storage.

The Real Type

Variables of type real have values which are elements of a subset of real numbers. In PASCAL-80, a value, r, of type real will satisfy the following:

```
      1.17*10E-38<=r<=3.40*10E38      (approximately)
   or -3.40*10E38<=r<=-1.17*10E-38    (approximately)
```

Further information concerning the characteristics of real values in PASCAL-80 can be obtained by consulting the Intel publication *8080/8085 Floating Point Arithmetic Library User's Manual,* Publication Number 9800452.

A variable of type real occupies four bytes of storage.

The Char Type

A variable of type char has values which are elements of the finite set of characters. In PASCAL-80, this set of values can be defined by:

```
TYPE char = chr(0)..chr(255)
```

The ASCII control and graphic coding is employed, and therefore so is the ASCII collating sequence.

A variable of type char occupies two bytes of storage, although the most significant byte is always zero.

The ARRAY Type

A variable of type ARRAY has values which are composed of a fixed number of components, each one of which is of the same type, called the base type. In PASCAL-80, the ARRAY type is as specified in Standard Pascal.

If the ARRAY declaration is prefixed by the reserved word PACKED, the compiler will attempt to minimize storage requirements by packing more than one component into each word. However, this packing is subject to the following restrictions:

1. Packing will never cross word boundaries. This implies that an array will be packed only if any value of the base type can be represented in one byte or less, such as boolean, char, or 0..73. Packing densities for various base type representations are as follows:

Bits to Represent Value	Components/Word
1	16
2	8
3-8	2
9 and up	1 (no packing)

2. An array will be packed if and only if the prefix PACKED appears immediately preceding the last occurrence of the symbol ARRAY. In particular,

```
a1: PACKED ARRAY[0..10] OF ARRAY[0..10] OF char;
```

will not be packed, while

```
a2: ARRAY[0..10] OF PACKED ARRAY[0..10] OF char;
a3: PACKED ARRAY[0..10] OF PACKED ARRAY[0..10] OF char;
a4: PACKED ARRAY[0..10,0..10] OF char;
```

will all be packed two characters per word.

3. A components of a PACKED ARRAY may not be passed as a variable parameter to a FUNCTION or PROCEDURE. It may, however, be passed as a value parameter.

The RECORD Type

A variable of type RECORD has values which are composed of a fixed number of fields, with no restriction on the type of each field. In PASCAL-80 the RECORD type is as specified in Standard Pascal.

If the RECORD declaration is prefixed by the reserved word PACKED, the compiler will attempt to minimize storage requirements by packing more than one field into each word. However, this packing is subject to the following restrictions:

1. Packing will occur only with adjacent fields of subrange or scalar type. Any field which is of a structured type (ARRAY, RECORD, SET, string, or FILE) will always begin on a word boundary.
2. Packing will never cross word boundaries. This implies that two or more fields will be packed into a word only if the total number of bits required for their combined representation is less than 16.
3. The prefix PACKED must appear before every occurrence of RECORD or ARRAY in order for the fields themselves to be packed.
4. A field of a PACKED RECORD may not be passed as a variable parameter to a FUNCTION or PROCEDURE. It may, however, be passed as a value parameter.

The String Type

PASCAL-80 contains the predeclared data type string. This type is similar to a PACKED ARRAY OF char, but it also incorporates a dynamic length byte. Therefore, even though the allocated length of a string remains fixed, its dynamic length can vary according to the assignments made to it. It is only the dynamic length of a string which is involved in any string operations. The allocated length of a string may be from 1 to 255 characters, 80 characters being the default length. In addition, the individual characters of a string may be referenced, using an index of 1 to the length of the string. The following examples illustrate the use of strings:

```
ch: char;
name: string[30];                  The variable name is of type
                                  string, with a maximum length
                                  of 30 characters
address: string;                   The variable address is
                                  of type string, with a maximum
                                  length of 80 characters

name := 'George Washington';
read(input,address);
ch := name[3];
```

The SET Type

In PASCAL-80, as in Standard Pascal, the SET type defines the set of all subsets of values of a base type, which must be either a scalar type or subrange type.

The maximum number of elements in a set in PASCAL-80 is 4080, each one of which must satisfy the following:

```
ord(<element>) >= 0
```

The FILE Type

As in Standard Pascal, a FILE is a structure consisting of a sequence of identical (i.e., of the same type) components. The type is distinguished from the ARRAY type in that the FILE type has a variable number of components.

In PASCAL-80, each structure of type FILE must be associated with an ISIS-II device or disk file, which are discussed in Chapter 2 of the ISIS-II User's Guide. This association is established by the **reset** or **rewrite** procedures, described below, and is terminated by the **close** procedure, also described below.

Note that every open file allows for update access i.e., each file is opened with an access mode value of 3. Standard Pascal does not allow a file to be opened for "input only" or "output only".

As in Standard Pascal, the two textfiles **input** and **output** are predeclared in the surrounding environment. These files are automatically associated with the ISIS-II system console devices :CI: and :CO:, respectively, and do not have to be opened by the Pascal program.

PASCAL-80 has two predeclared file types which are extensions to Standard Pascal: Untyped files and INTERACTIVE files.

Untyped Files

Untyped files were introduced so that a program could perform efficient, unstructured block transfers to and from memory. These transfers are performed with the procedures blockread, blockwrite, bufferread, and bufferwrite.

The following example program will copy file SSS to file DDD, using an 8K memory buffer:

```
program fcopy;
const size = 8192;
var s,d: file;                     s and d are untyped files
    buffer: packed array[1..size] of char;
    length: integer;
begin
   reset(s,'SSS');                 open SSS for reading
   rewrite(d,'DDD');               and create DDD if necessary
   length := bufferread(s,buffer,size);   read in the first chunk
   while length > 0 do
      begin
         length := bufferwrite(d,buffer,size);   write full chunk
         length := bufferread(s,buffer,size);    and read the next
      end
   if length < 0 then              write out the last bit of the file
      length := bufferwrite(d,buffer,-length)
   else
      writeln(output,'Error in reading file');
   close(s); close(d);
end.
```

INTERACTIVE Files

In the PASCAL User Manual, the statement read(f,ch) is defined to be

```
ch := f ;get(f)
```

if f is of type TEXT (FILE of char). This definition implies that a get(f) is executed upon opening the file, so that the first assignment from the buffer variable will be valid. This initial get(f) is a cumbersome constraint when the file is opened to an interactive terminal, since a program will hang waiting for that initial character. In order to solve this problem, the file type INTERACTIVE exists. A file of type INTERACTIVE is similar to a file of type TEXT in all respects save that read(f,ch) is defined to be

```
get(f); ch := f ;
```

and a get(f) is not performed when the file is opened. The predeclared file **input** is of type INTERACTIVE.

The POINTER Type

In PASCAL-80, as in Standard Pascal, the POINTER type defines a set of values pointing to elements of a given type.

PROCEDUREs and FUNCTIONs

As in Standard Pascal, PASCAL-80 allows for associating an identifier with a program part, and hence defining named PROCEDUREs and FUNCTIONs. The syntax for these declarations is the same as in Standard Pascal, except in one respect: In PASCAL-80, procedures and functions may not be used as arguments to other procedures or functions.

SEGMENT Procedures

In order to allow for the execution of very large programs which cannot fit into main memory, PASCAL-80 allows these programs to be partitioned into SEGMENT PROCEDUREs or SEGMENT FUNCTIONs. This language construct allows for dynamic overlaying, in that a SEGMENT PROCEDURE (FUNCTION) is resident in memory only when it is a part of the dynamic execution thread. Once a SEGMENT PROCEDURE (FUNCTION) exits, the portion of memory which it occupied is now free for some other overlay.

A SEGMENT is declared as follows:

```
SEGMENT PROCEDURE initialize(firsttime: boolean);
     .....
```

and

```
SEGMENT FUNCTION getcharacter: char;
     .....
```

where "....." denotes the body of the PROCEDURE (FUNCTION) which has the same syntax as non-overlay program parts.

The main body of program is always resident, as is every procedure which is not a SEGMENT. All SEGMENT declarations must occur before any of the resident procedure declarations or the main body of the program.

CHAPTER 3
PREDECLARED PROCEDURES

As in all implementations of the Pascal programming language, PASCAL-80 contains a number of predeclared procedures and functions. These include the standard procedures as defined in the PASCAL User Manual, plus additional procedures which are extensions to Standard Pascal. These predeclared procedures and functions are declared in the Run-Time Environment surrounding each Pascal program, and may be redeclared with no conflict.

The predeclared procedures and functions include generally useful utility routines, data structure access routines, and routines designed to allow PASCAL-80 programs access to the powerful system features of ISIS-II. The following discussion assumes that the reader is familiar with the characteristics of ISIS-II, as described in the Intel publication *ISIS-II User's Guide,* Publication Number 9800306.

File Manipulation Procedures

reset(f)

Resets the current file position to the beginning of the file f, and positions the file pointers to the first record in the file. If f is not an INTERACTIVE file, a get(f) is performed. The function eof(f) becomes true if f is empty, otherwise eof(f) becomes false. See figures 3-1 and 3-3 for examples.

reset(f,<string>)

Opens an existing ISIS-II file with the name <string>, associates f with that file, and then performs reset(f). See figure 3-2 for an example.

rewrite(f,<string>)

Creates an ISIS-II file with the name <string>, associates f with that file, and sets the file pointers to the beginning of the file. See figure 3-1 for an example.

close(f)

Closes the file associated with f and removes that association. See figure 3-1 for an example.

put(f)

Writes the value of the buffer variable f↑ to the file at the current file position and advances the position to the next component. The procedure put(f) is defined only for typed files. See figure 3-1 for an example.

get(f)

Assigns the value of the current component of the file f to the buffer variable f↑, and advances the file pointers to point to the next component. If no current component exists, eof(f) becomes true, and the value of f↑ is undefined. The function eof(f) must be false upon entry, and f must be a typed file. See figures 3-1 and 3-2 for examples.

```
{
This program illustrates the use of the predeclared procedures
reset, rewrite, get, put, and close:
}

program threel;

  var
    i: integer;
    datafile: file of integer;

  begin  { 3-1 }
    rewrite(datafile,':F1:DDATA.DAT');  { create DDATA.DAT on drive 1 }
    for i := 0 to 10 do                 { write 0..10 onto the file }
      begin
        datafile^ := i;
        put(datafile);
      end;
    reset(datafile);                    { re-position file to beginning }
    repeat                              { read in and display the integers }
      writeln(datafile^:5);
      get(datafile);
    until eof(datafile);
    close(datafile);                    { and then close the file }
  end.  { 3-1 }
```

```
Executing this program will result in the following output:

    0
    1
    2
    3
    4
    5
    6
    7
    8
    9
   10
```

Figure 3-1. Examples of Predeclared Procedures reset, rewrite, get, put, and close

seek(f,<integer>)

Positions the file pointers to the <integer>th component of the file f. See figure 3-2 for an example.

read(f,v1,...,vj)

This procedure may be used only on TEXT or INTERACTIVE files. If f is omitted, input is assumed. Note that this construct is equivalent to:

```
read(f,v1); ... ; read(f,vj)
```

If vi is of type string, read(f,vi) will read up to the end of the line and set eoln(f) true. See figure 3-3 for an example.

readln(f,v1,...,vj)

This construct is equivalent to the sequence:

```
read(f,v1,...,vj); readln(f)
```

where readln(f) is used to read to, and subsequently skip past, the end of the current line. If the file is not exhausted, eof(f) and eoln(f) are set false. See figure 3-3 for an example.

```
{
This program uses the data file created in program threel to
illustrate the use of the predeclared procedures
reset, seek, and get:
}

program three2;

  var
    i: integer;
    datafile: file of integer;

  begin  { 3-2 }
    reset(datafile,':F1:DDATA.DAT');  { open the file DDATA.DAT on drive 1 }
    for i := 10 downto 6 do
      begin
        seek(datafile,i);                       { seek to a record }
        get(datafile);                          { get its value }
        write(output,'Record number:',i:3);     { and display the data }
        write(output,'   Record value:',datafile^:3);
        writeln(output);
      end;
    close(datafile);
  end.  { 3-2 }
```

```
Executing this program will result in the following output:

Record number: 10   Record value: 10
Record number:  9   Record value:  9
Record number:  8   Record value:  8
Record number:  7   Record value:  7
Record number:  6   Record value:  6
```

Figure 3-2. Examples of Predeclared Procedures reset, seek, and get

```
{
This program illustrates the use of the predeclared procedures
read and readln:
}

program three3;

  var
    a,b,c,y,z: char;
    datafile: text;

  begin  { 3-3 }
    rewrite(datafile,':F1:TDATA.DAT');  { create a temporary file on drive 1 }
    writeln(datafile,'AB');             { write a two character line }
    writeln(datafile,'YZ');             { and a two character line }
    reset(datafile);                    { re-position file to the beginning }
    a := '1';                           { initialize variables }
    b := '2';
    c := '3';
    y := '8';
    z := '9';
    read(datafile,a,b,c,y,z);           { c := ' ' since at end-of-line }
    writeln(a,b,c,y,z);                 { display the values }
    reset(datafile);
    a := '1';
    b := '2';
    c := '3';
    y := '8';
    z := '9';
    readln(datafile,a);                 { read the 'A'; skip to the next line }
    readln(datafile,y);                 { read the 'Y' and skip to eof }
    writeln(a,b,c,y,z);
    close(datafile);
 end.  { 3-3 }
```

```
Executing this program will result in the following output:

AB YZ
A23Y9
```

Figure 3-3. Examples of Predeclared Procedures read and readln

write(f,v1,...,vj)

This procedure may be used only with TEXT or INTERACTIVE files. If f is omitted, output is assumed. Note that this construct is equivalent to

```
write(f,v1); ...; write(f,vj).
```

See figures 3-2 and 3-3 for examples.

writeln(f,vl,...,vj)

This construct is equivalent to the sequence:

```
write(f,v1); ...; write(f,vj)
```

where writeln(f) writes an end of line character of the file f. See figures 3-2 and 3-3 for examples.

page(f)

Causes a form-feed character to be written to the TEXT or INTERACTIVE file f.

File Manipulation Functions

bufferread (f,<array>,<length>[,<break-char>]): integer
bufferwrite(f,<array>,<length>[,<break-char>]): integer

These functions are used to read and write arbitrary length buffers from/to an untyped file f. The maximum number of bytes which will be transferred is given by <length>, so <array>, a PACKED ARRAY OF char, should be at least this long. If <break-char>, an integer expression 0..255, is specified, bytes will be transferred up to and including the break byte. The values of these functions are as follows, where <length> signifies the number of bytes transferred:

```
buffer<r/w>(f,a,n) = n if no eof, else -<length>

buffer<r/w>(f,a,n,b) = <length> if break on byte
        value b else -<length> if eof or no break.
```

blockread (f,<array>,<blocks>[,<start-block>]): integer
blockwrite(f,<array>,<blocks>[,<start-block>]): integer

These functions are used to read and write 512 byte blocks from/to an untyped file f. Both functions return the number of blocks actually transferred. The length of <array>, a PACKED ARRAY OF char, should be a multiple of 512 bytes, since no array bounds checking is done. If <start-block> is specified, a seek(f,512*<start-block>) will be performed to that position in the file; otherwise, the transfer will begin at the current file position.

eof(f): boolean

Returns true if file f is positioned at end of file, else returns false. If f is a disk file, eof(f) will become true when f is exhausted; i.e. when the file MARKER is equal to the file LENGTH. If f is associated with :CI:, pressing CONTROL/Z in lieu of requested input data will cause eof(f) to become true.

eoln(f): boolean

Returns true if the TEXT or INTERACTIVE file f is positioned at an end-of-line character, else returns false.

ioresult: integer

Returns the error code of the last I/O operation. If no error was detected, the value is 0.

Dynamic Allocation Procedures

new(p)

Allocates a new variable t and assigns the address of t to the pointer variable p, where p is defined as:

```
VAR p:  t;
```

If the type of t is a record type with variants, new(p) allocates a storage area large enough to accommodate the largest variant. See figure 3-4 for an example.

new(p,t1,...,tj)

Allocates a variable of the appropriate size for the variant with tag field values equal to the constants t1,...,tj and assigns the address of that variable to the pointer variable p.

mark(p)

Assigns the address of the current top of the HEAP to the pointer variable p. See figure 3-4 for an example.

release(p)

Sets the top of the HEAP to the value of the pointer variable p. See figure 3-4 for an example.

memavail: integer

Returns the arithmetic difference between the current bottom of the execution stack and the top of the heap, in 16-bit words.

Arithmetic Functions

abs(x): <type of x>

Computes the absolute value of x. The type of x must be either integer or real, and the type of the function is the type of x.

sqr(x): <type of x>

Computes x*x. The type of x must be either integer or real, and the type of the function is the type of x.

```
{
This program illustrates the use of the predeclared procedures
new, mark, and release:
}

program three4;

  type
    intarray = array[0..10] of integer;

  var
    i: integer;
    htop, htopl: ^integer;
    buffer: ^intarray;

  begin
    mark(htop);                      { record the top of the HEAP }
    new(buffer);                     { allocate an array of integers from
                                       the top of the HEAP. This moves the
                                       top of the HEAP upwards by 22 bytes. }
    mark(htopl);                     { record the new HEAP top }
    writeln(ord(htopl) - ord(htop)); { display the amount of allocated space }

    for i:=0 to 10 do                { set the values of the array }
      buffer^[i] := i*i;

      { process the array }

    release(htop);                   { set the top of the HEAP back to its
                                       original value, which was saved in
                                       htop. }

    { The pointer variable 'buffer' now points to locations above the
      valid area of the HEAP, and must not be used until, using the
      'new' procedure again, it points to a valid array. }

  end.  { 3-4 }
```

Executing this program will result in the following output:

```
        22
```

Figure 3-4. Examples of Predeclared Procedures new, mark, and release

sin(x): real
cos(x): real
log(x): real
exp(x): real
ln(x): real
sqrt(x): real
arctan(x): real

These functions all require x to be either integer or real, and all return a result of type real.

trunc(x): integer

Returns the whole part of the real argument x. The fractional part is discarded, so that trunc(5.78)=5, and trunc(−2.9)=−2.

round(x): integer

Returns the rounded integer of the real argument x. This function is defined as follows:

round(x) = trunc(x + 0.5) if x>=0
round(x) = trunc(x−0.5) if x<0

pwroften(x): real

Returns 10 raised to the xth power, where x is an integer such that 0<=x<=38.

Predicates

odd(x): boolean

Returns true if x is an odd integer, otherwise returns false; x must be of type integer.

Transfer Functions

ord(x): integer

Returns the ordinal number of the value of x in the set defined by the type of x. Note that x must be of a scalar type. See figure 3-5 for an example.

chr(x): char

Returns the character whose ordinal number is the value of x, if it exists; x must be of integer type. See figure 3-5 for an example.

Miscellaneous Routines

succ(x): <type of x>

Returns the successor value of x, if it exists; x must be of a scalar or subrange type.

pred(x): <type of x>

Returns the predecessor value of x, if it exists; x must be of a scalar or subrange type.

```
{
This program illustrates the use of the predeclared procedures
ord and chr:
}

program three5;

  type
    colors = (red,green,yellow,blue);

  begin
    writeln(ord(red):3,ord(green):3,ord(yellow):3,ord(blue):3);
    writeln(chr(ord('A')));
  end.  { 3-5 }


Executing this program will result in the following output:

  0  1  2  3
A
```

Figure 3-5. Examples of Predeclared Procedures ord and chr

sizeof(<variable> | <type identifier>): integer

Returns the number of bytes that would be or are allocated for the argument.

exit(<procedure/function identifier>)

Causes an exit from the named procedure or function, which should be part of the dynamic execution thread. If the identifier is the name of the program itself, then execution will terminate and control will return to the command handler of the PASCAL-80 Run-Time System. See figure 3-6 for an example.

```
{
This program illustrates the use of the predeclared procedure
exit:
}


program three6;

  var
    i: integer;

  procedure alpha;
    forward;
  procedure beta;
    forward;
  procedure gamma;
    forward;

  procedure alpha;
    begin
      beta;
      writeln('Exit alpha');
    end;  { alpha }

  procedure beta;
    begin
      gamma;
      writeln('Exit beta');
    end;  { beta }

  procedure gamma;
    begin
      write('Please enter a digit: ');
      readln(i);
      writeln;
      writeln('Thank you');
      case i of
        1: exit(three6);
        2: exit(alpha);
        3: exit(beta);
        4: exit(gamma);
      end;  { case }
      writeln('Exit gamma');
    end;  { gamma }

  begin
    alpha;
    writeln('Exit program');
  end.  { 3-6 }


Executing this program will result in the following output
  sequences, depending upon the value of <n>:

Please enter a digit: <n>      { <n> is entered by the user }

Thank you      Thank you     Thank you     Thank you     Thank you
               Exit program  Exit alpha    Exit beta     Exit gamma
                             Exit program  Exit alpha    Exit beta
                                           Exit program  Exit alpha
                                                         Exit program

 {<n> = 1}      {<n> = 2}     {<n> = 3}     {<n> = 4}     {<n> >= 5 or
                                                           <n> <= 0}
```

Figure 3-6. Example of Predeclared Procedure exit

gotoxy(<column>,<row>)

Positions the CRT cursor to the specified <column> and <row>. Both arguments must be integers. Position (0,0) is the top left corner of the screen.

setpointer(p,v)

Places the address of the variable v into the pointer variable p.

portinput(p,v)

Inputs a value from the 8080/8085 I/O port p and set variable v to that value.

portoutput(p,e)

Outputs the value of the integer expression e to the 8080/8085 I/O port p.

errorset(p)

Causes p to become the current error procedure. The procedure p must be a parameterless procedure declared at the global level of the Pascal program. The errorset(p) procedure may be executed as often as desired; each invocation causes the previous error procedure to be overridden by the new procedure specified by errorset.

String Manipulation Routines

length(s): integer

Returns the current value of the dynamic length of the string s. See figure 3-7 for an example.

pos(<pattern>,s): integer

Returns the position in the string s of the first occurrence of the string <pattern>. If <pattern> is not a substring of s, 0 is returned. The position returned is the position of the first character of the match. See figure 3-7 for an example.

concat(s1,...,sj): string

Returns a string which is the concatenation of the strings s1 to sj. There is no restriction on the number j except that the resultant string be less than 256 characters. See figure 3-7 for an example.

copy(s,<index>,<length>): string

Returns a substring of s, starting at position <index>, of <length> characters. See figure 3-7 for an example.

insert(s1,s2,<index>)

Inserts the string s1 into the string s2, starting at the <index>th character in s2. See figure 3-7 for an example.

```
{
This program illustrates the use of the string manipulation routines
length, pos, concat, copy, insert, and delete:
}


program three7;

  var
    sa,sb,sc,sd: string[70];

  begin
    sa := 'First STRING to be defined';
    sb := 'Second STRING we are defining';
    sc := ' in this example';
    sd := '';                    { null string }
    writeln(length(sa):3,length(sd):3);
    writeln(pos('nd ',sb):3, pos('XXX',sc):3);
    writeln(concat(sb,sc));
    writeln(copy(sa,7,3));
    insert('Pascal ',sb,8);
    writeln(sb);
    delete(sa,14,6);
    writeln(sa);
  end.  { 3-7 }
```

Executing this program results in the following output:

```
 26  0
  5  0
Second STRING we are defining in this example
STR
Second Pascal STRING we are defining
First STRING defined
```

Figure 3-7. Examples of String Manipulation Routines length, pos, concat, copy, insert, and delete

delete(s,<index>,<length>)

Deletes <length> characters from the string s, starting at position <index>. See figure 3-7 for an example.

Character Array Manipulation Routines

These routines operate on PACKED ARRAY OF char structures. No range checking is performed, so they should be used with some care.

scan(<length>,<partial expression>,a): integer

Scans the ARRAY a for a character which satisfies the partial expression, which must be of the form:

"<>" or "=" followed by <character expression>

where <character expression> is an expression which evaluates to a character value. The ARRAY a may be subscripted to denote a starting point; otherwise the scan starts at the beginning of a. The number of characters scanned is returned. If the first character satisfies the <partial expression>, 0 is returned; if no character is satisfactory, <length> is returned. If <length> is negative, the array is scanned backwards and the value returned will be <=0. See figure 3-8 for an example.

moveleft (<source>,<dest>,<length>)
moveright(<source>,<dest>,<length>)

Both of these routines move <length> bytes from ARRAY <source> to ARRAY <dest>. Either or both <source> and <dest> may be subscripted to denote a starting position other than the first character. The moveleft routine starts at the specified left end of both arrays and copies moving right. The moveright routine starts at the right end of both arrays and copies bytes moving left. See figure 3-8 for an example.

fillchar(a,<length>,<character>)

Fills the ARRAY a with <length> <character>'s. The ARRAY may be subscripted to denote a starting position other than the first character. See figure 3-8 for an example.

```
{
This program illustrates the use of the character array routines
scan, moveleft, moveright, and fillchar:
}


program three8;

  var
    ca,cb: packed array[0..9] of char;
    sa: string[30];

  begin
    ca := '0123456789';
    writeln(scan(10,='4',ca):3, scan(10,='4',ca[2]):4);
    writeln(scan(10,='X',ca):3, scan(5, ='X',ca):4);
    writeln(scan(-10,='8',ca[9]):3, scan(-10,='X',ca[9]):4);

    moveleft(ca,cb,sizeof(ca));      { copy ca to cb both ways }
    writeln(cb);
    moveright(ca,cb,sizeof(ca));
    writeln(cb);
    moveleft(ca,ca[1],9);            { the incorrct way to move
                                       characters up by one position }
    writeln(ca);
    moveright(cb,cb[1],9);         { the correct way }
    writeln(cb);

    sa := 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';
    writeln(sa);
    fillchar(sa[1],25,'0');          { a string is a packed array of char }
    writeln(sa);
  end.  { 3-8 }
```

Executing this program results in the following output:

```
  4   2
 10   5
 -1 -10
0123456789
0123456789
0000000000
0012345678
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0000000000000000000000000Z
```

Figure 3-8. Examples of Character Array Routines scan, moveleft, moveright, and fillchar

CHAPTER 4
OPERATING INSTRUCTIONS

The PASCAL-80 System is composed of a Run-Time System (RTS) and a compiler which itself executes on the Run-Time System. The PASCAL-80 compiler converts PASCAL source code into an intermediate code which is then interpreted by the Run-Time System.

The ISIS-II text editor is used to create the Pascal source file(s). The PASCAL-80 compiler is then used to compile the source programs so that they may be executed on the RTS. A listing of the PASCAL-80 program is provided by the compiler during compilation. Any error messages are included in this listing.

Getting Started with PASCAL-80

Each user is provided with all the software that makes up the PASCAL-80 development system. This software is contained on each of two (2) diskettes (single and double density); be sure to use the appropriate diskette on your disk hardware. The directory listing of each of these diskettes can be found in Appendix K.

Release Disks

Each PASCAL-80 Release Disk contains the software listed below:

Compiler	RTS	Utilities	Libraries	Example	Figures
COMP.COD	PASCAL	JOIN.COD	P80RUN.LIB	EX.PAS	FIG31.PAS
	PASCAL.RES	GENOBJ.COD	P80ISS.LIB	BUFFER.PAS	FIG32.PAS
			P80RAR.LIB	SEEKEX.PAS	FIG33.PAS
			P80EXT.LIB	ERROR.PAS	FIG34.PAS
			P80ISS.PLB	PEOPLE.DAT	FIG35.PAS
					FIG36.PAS
					FIG37.PAS

Initial Steps

It is highly recommended that you perform the following steps to get started.

1. Initialize a blank diskette as an ISIS-II system diskette using the IDISK command.
2. Copy all files from your PASCAL-80 Release Disk onto the ISIS-II system diskette to create a PASCAL-80 system diskette that you can load from drive 0.
3. If you run out of space on the ISIS-II system diskette, note that in order to run PASCAL-80 you need only the following files:

   ISIS.DIR
   ISIS.MAP
   ISIS.TO
   ISIS.LAB
   ISIS.BIN
   ISIS.CLI

4. For most applications, the following files on the ISIS-II system diskette are also useful (but not required to run PASCAL-80):

```
DIR
COPY
DELETE
RENAME
ATTRIB
SUBMIT
```

5. It is a good practice and highly recommended that the files containing PASCAL be write-protected. Using the ISIS-II ATTRIBUTE command, change the attributes of the following files to W1:

```
PASCAL, PASCAL.RES and COMP.COD
```

Procedures for performing these functions are given in the *ISIS-II User's Guide,* Order Number 9800306.

Compiler

The PASCAL-80 Compiler is overlaid and loads its overlays from the drive containing COMP.COD.

Run-Time System

The RTS executes the intermediate code output from the compiler. Its code is contained in the files PASCAL and PASCAL.RES.

Libraries

P80RUN.LIB, P80ISS.LIB, and P80RAR.LIB are ISIS-II object module libraries used in the generation of Load and Go versions of a Pascal program. They are discussed more fully in Chapter 6.

Example Programs

EX.PAS, BUFFER.PAS, ERROR.PAS and SEEKEX.PAS are simple demonstration programs, supplied in source form, which show many of the facilities present in PASCAL-80, and which can also be used by the newcomer to familiarize himself with the system. (SEEKEX.PAS is somewhat more complex than the other programs.)

Figure Programs

Each one of the example programs in Chapter 3 is supplied in source form, so that they may be compiled and executed by the user. These files are named FIG3n.PAS, where n is a digit from 1 to 7.

Run-Time System Memory Layout

The PASCAL-80 Run-Time System is composed of an interpreter and a number of predeclared procedures and functions written in Pascal. During interpreter execution, there are two major dynamic data structures: the STACK and the HEAP. The STACK contains all P-code and data frames; all dynamic variables are allocated from the HEAP. The layout of main memory is illustrated in figure 4-1.

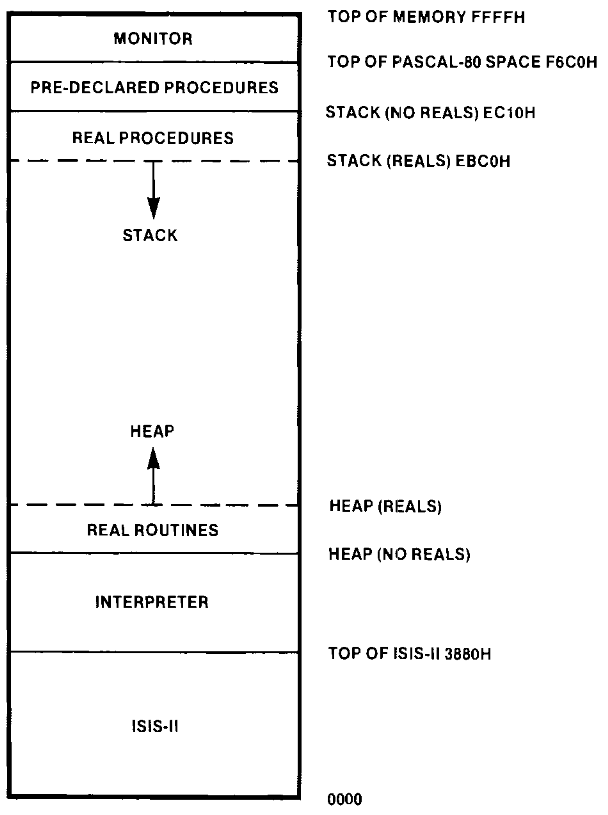


Figure 4-1. Main Memory Layout for a 64KB System 1015-2

Operating the PASCAL-80 System

Invoking PASCAL-80

To invoke the PASCAL-80 System from ISIS-II, type the following:

```
-PASCAL [<directive>] [<code-filename> [<options>]]
```

The directive is an optional field which, when present, causes the PASCAL-80 System to be loaded without the code for real numbers. The form of this option is:

```
-R
```

This option is used to increase the memory available to programs which do not use real numbers. With the -R option specified, approximately 2800 additional bytes are available. For example:

```
-PASCAL -R
PASCAL-80 Vv.r [No reals]
>
```

The code-filename is also an optional field which, when present, allows for automatic Pascal program execution. This feature is described in the Automatic Program Execution section.

If neither of the optional fields is present, PASCAL-80 responds with:

```
PASCAL-80 Vv.r
>
```

Command Line Syntax for PASCAL-80

When the PASCAL-80 Run-Time System is invoked, it responds as specified above, and displays its prompt character ">" to indicate that it is waiting for a command which is defined as follows:

```
<command>           ::= <file-command> | <direct-command>
<file-command>      ::= <code-filename> [<options>]
<direct-command>    ::= TRACEON | TRACEOFF | QUIT |
                        SIZEON | SIZEOFF | STATS
```

The code-filename specifies a compiled Pascal program which will be loaded by the RTS and executed. The meaning of the <options> field is determined by the program being executed. It allows for the specification of data files, execution options, and directives at the time of invoking the program.

The next two commands, TRACEON and TRACEOFF, are used to control the tracing facility of the PASCAL-80 System.

The QUIT command returns control to the ISIS-II Command Line Interpreter.

The SIZEON and SIZEOFF commands are used to control the run-time monitoring of the STACK size and HEAP size.

The STATS command causes certain run-time statistics to be displayed.

Examples of valid commands are as follows:

```
>COMP TEST.PAS
>TEST
>TRACEON
>TRACEOFF
>QUIT
```

The first command will compile the Pascal source program file TEST.PAS. The second command will execute the Pascal object program TEST.COD, which the previous compilation just generated.

The third command will turn on the TRACE flag, so that the line number of each executed program statement will be displayed when a user program is next executed. The fourth command turns off the TRACE flag, while the fifth command returns control to ISIS-II.

Code Filename Specification

A code-filename may be expressed in the following ways:

Name Typed	Name Used
<filename>.<ext>	<filename>.<ext>
<filename>	<filename>.COD
<filename>.	<filename>

Command Line Options

When the command line specifies a Pascal code file to be executed, that codefilename may be followed by an arbitrary <options> field. This <options> field specifies run-time parameters to the program being invoked, and may be input by that program using the standard "read" and "readln" procedures. When a program starts executing, the function eoln(input) will reflect the presence of an <options> field. It will be false if options are present, and true otherwise.

The following Pascal statements can be used to collect the options from the command line (if any) and place them in the string variable "options":

```
if eoln(input) then
  options := ''
else
  readln(input,options);
```

The program may then inspect the value of "options" and perform the appropriate actions.

Program Tracing Facility

The PASCAL-80 System incorporates a program tracing facility which allows for selectively monitoring the execution of a Pascal program. When the TRACE flag is set, the line number of each program statement being executed is output to :CO:, enclosed in brackets ("[" and "]"). Using this information, together with the compiler listing which associates line numbers with program statements, a programmer may more easily determine what the program is doing.

As an example of the tracing output, if the program were to ask for a customer name:

```
[137][138]Customer name: [139]
```

might be displayed if the TRACE flag were set, while:

```
Customer name:
```

would only be displayed if the TRACE flag were not set.

The TRACE Flag

The TRACE flag may be manipulated in two ways. The TRACEON command will set the flag, and the TRACEOFF command will reset the flag. Additionally, pressing the INTERRUPT 4 switch on the Intellec system front panel will cause the TRACE flag to toggle. In particular, every time the INTERRUPT 4 switch is pressed, the TRACE flag will be set if it was reset, or will be reset if it was set. This allows for selectively tracing only those portions of a program which are in question, and executing the remainder of the program normally.

Tracing Instructions

In order for the PASCAL-80 System to trace a program, or portions of a program, tracing instructions must be present in the compiled code. The PASCAL-80 compiler normally inserts these instructions in the object code, but they may be omitted using the embedded compiler directive {$T-}. When the compiler encounters this directive, it will not issue tracing instructions in any subsequent code which is generated. This situation will persist until a {$T+} directive is encountered, at which time tracing instructions will begin to be generated again. Therefore, using these directives, an entire Pascal program, or just selected portions, may contain tracing instructions.

Run-Time Monitoring

The PASCAL-80 Run-Time System has the capability to continuously monitor the dynamic sizes of both the STACK and the HEAP. Since a performance degradation occurs when this monitoring is in effect, the user is able to turn it on and off with the SIZEON and SIZEOFF commands. When size monitoring is in effect, the system keeps track of three values as they are varied by the running user program:

- The maximum size of the STACK at any time.
- The maximum size of the HEAP at any time.
- The maximum combined size of the STACK and HEAP at any time.

These values are initialized to zero when a program commences execution, and are saved when it returns to the Pascal Command Line Interpreter. The STACK size value does not include the space consumed by the resident code of the Pascal program, but does include the space occupied by each data frame, including the program's global data frame, and also the length of any SEGMENT procedures.

The STATS command will display these saved values. They are useful when generating a Load and Go version of a Pascal program, or when constructing an RMX/80 PASCAL task, since the combined STACK and HEAP value is the maximum free space needed by the program. Note, however, that this free space value is valid only when the program has execution characteristics similar to those during the actual size monitoring, and should not be accepted as the maximum free space the program will ever need. In particular, if the program has declared procedures or functions which are recursive, the number of levels of recursion is one factor contributing to the size of the STACK, and therefore each invocation of the program will require a different amount of free space depending upon the recursion level reached. Also, the particular sequence of procedure calls and returns, as well as the pattern of dynamic variable allocation on the HEAP, determine the free space used during any specific invocation of a program.

Displaying the Statistics

The execution of the STATS command will cause the value of certain run-time variables to be displayed. In addition to the size monitoring variables described above, the current size of the free space (between the top of the HEAP and the bottom of the STACK) is also displayed. The format of the STATS command is as follows:

```
<STATS
Interpreter Version: Vx.y
Available Free Space: >size-1>
Maximum HEAP Size: <size-2>
Maximum STACK Size: <size-3>
Maximum Combined Size: <size-4>
Run Time Monitoring: <flag-value-1>
Run Time Tracing: <flag-value-2>
```

where:

x.y	The version and revision of the Pascal Interpreter itself.
<size-1>	The number of bytes available for the code and data of a user program.

<size-2>	
<size-3>	
<size-4>	The current values of the size monitoring variables.
<flag-value-1>	
<flag-value-2>	The state of the associated Run Time parameters. The possible values of these parameters are On or Off.

Automatic Program Execution

If a code-file is specified on the ISIS-II command line when invoking PASCAL-80, the PASCAL-80 System is loaded and code-filename is executed with no further user input. For example, typing

```
-PASCAL COMP EX.PAS
```

would result in the following:

```
PASCAL-80 Vv.r


PASCAL-80 Compiler Vv.r
COMPILING EX.PAS

Symbol table space remaining : 12234 bytes
37 lines compiled

PASCAL-80 Vv.r

>
```

Interrupting Program Execution

INTERRUPT switch 3 on the Intellec system front panel causes the currently executing Pascal program to cease execution, and causes control to be returned to the PASCAL-80 System command handler. An error message is displayed, along with the Segment, Procedure, and Instruction where execution was suspended.

Operating the PASCAL-80 Compiler

Command Line Syntax for the Compiler

The command line syntax for compiling a Pascal program is as follows:

```
COMP <filename> [<directives>]
```

where filename is the name of the source code file. If the filename is not given, an error message is printed and control is returned to the PASCAL-80 system.

The source filename may be specified in the following ways:

Name Specified	Name Used
<filename>.<ext>	<filename>.<ext>
<filename>	<filename>.PAS
<filename>.	<filename>

The <directives> field is an optional sequence of compiler directives (see below). Each directive must be separated from the preceding directive by one or more spaces. Where directives have brackets, the left hand bracket may occur zero, one, or more spaces after the body of the directive. To terminate the sequence, press return.

If a COMP command is longer than one line on your console, (which must not be greater than 122 characters), you can continue it by entering an ampersand (&) before the carriage return. The ampersand cannot appear within a directive name or file name.

CAUTION

COMP creates a work file named P80WRK.TMP on the diskette to which the output code file is directed. If you have a file by this name on the output diskette, it will be destroyed.

Compiler Directives

The PASCAL-80 System compiler recognizes various directives which are used to control various phases and details of the compilation process. These directives are partitioned into two classes. The first class is the command line directives, which are specified on the command line after the name of the PASCAL-80 source file to be compiled. The second class is the embedded directives, which are embedded in the text of the program being compiled. The command line directives are described below; a summary of all directives is in Appendix C.

Compiler Command Line Directives

A description of the available command line directives follows:

NOLIST

No list file is produced.

NOCODE

No intermediate code file is produced.

ERRLIST

The listing is limited to those lines containing syntax errors.

LIST (external-file-name)

Specifies the file to which the listing is to be directed. The default is: <source-filename>.LST.

CODE (external-file-name)

Specifies the file to which the code is to be directed. The default is: <source-filename>.COD.

NOECHO

Error lines are echoed on the console unless this directive is specified.

GLOBAL(external-file-name)

Specifies the global symbol table file when separately compiling a partitioned Pascal program. The default is: <source-filename>.SYM.

WORKFILE(<device>)

Specifies which diskette device is to be used for the compiler's workfile, P80WRK.TMP. The default value is the device of the output code file.

DATE(<date>)

Specifies the date to be included in the page heading of the compiler listing. The <date> parameter is any sequence of nine or fewer characters not containing parentheses. The default value is all space.

ETAB(external-file-name)

Specifies the external table file when compiling a Pascal program with external references. The default is: <source-filename>.ERT.

NOSTATISTICS

Specifies that the compiler should not accumulate or list the procedure data frame and parameter sizes.

Excluded Combinations

Certain of these directives may not be used in combinations. The following table shows which directives are excluded if the directive in the left-hand column is specified.

Directive	Excluded Directives
NOLIST	LIST ERRLIST
NOCODE	CODE

Summary of Information on CRT

The general format of the basic command line is:

```
COMP filename [directives]
```

The compiler will reply with:

```
PASCAL-80 Compiler Vv.r
```

Where v is the version number and r is the release number.

Each directive is then acknowledged by the compiler on a separate line, and is either ACCEPTED or REJECTED. After all the directives have been acknowledged and none were rejected, the compiler opens its files and starts to compile. If any directive was rejected it will display:

```
** Compilation Terminated **
```

and control is returned to the PASCAL-80 System.

If all directives were accepted it will display the message:

```
COMPILING filename
```

If any file fails to open correctly, the compiler will display:

```
filename failed to open
```

The compilation will be terminated, returning control to the PASCAL-80 System.

When the compilation is complete, the compiler displays the message:

```
Symbol table space remaining : nnnnn bytes
mmmmm lines compiled
```

where nnnnn is the number of bytes of memory that were not used during compilation of the program. This tells you how much more symbol table space your program can use. The mmmmm is the number of lines in the program.

Compiler Listing Format

The general layout of the list file is as follows:

```
PASCAL-80 Compiler Vv.r                  filename        date        page: nnn

   Line    Seg    Proc    Lev    Disp

   nnnn    nn     nnn     n      nnnn    statement 1

    .

    .

    .

   nnnn    nn     nnn     n      nnnn    statement n
```

The first two lines are title information and are repeated for each page. Line is the line number, Seg is the segment number, Proc is the procedure number, Lev is the current level of nesting, and Disp is the displacement of the variable or statement in the current procedure. These numbers are useful in debugging since, when a run-time error occurs, the segment number, procedure number and instruction displacement are displayed. See Appendix E, Run-Time Errors, for more information. Refer to Chapter 5 for examples of program listing.

Initial Compilations

When developing a PASCAL-80 program, it is often desired to perform a quick syntax and semantic check of the program. Compiling a program with NOSTATISTICS, NOLIST and NOCODE specified on the command line will perform these checks in the minimum possible time.

CHAPTER 5
EXAMPLE PROGRAMS

The Pascal programs presented in this chapter are designed to illustrate both the basic characteristics and some of the language extensions in PASCAL-80.

Compiling the Example Programs

Compile all the demonstration programs on the release diskette. These programs have the extension .PAS.

For example:

```
-PASCAL

PASCAL-80 Vv.r

>COMP EX.PAS

PASCAL-80 Compiler Vv.r
COMPILING EX.PAS

Symbol table space remaining : 12234 bytes
37 lines compiled

PASCAL-80 Vv.r

>
```

NOTE

All of the examples in this manual assume that a PASCAL-80 system diskette is created by following the initial steps in Chapter 4, and that this Pascal software diskette is loaded in drive 0. If the diskette was loaded in drive 1, the first line in the above example would be:

```
:F1:PASCAL
```

the third line would be:

```
:F1:COMP :F1:EX.PAS
```

and the other examples in this manual would be modified accordingly.

To return to ISIS-II, type QUIT.

For example:

```
PASCAL-80 Vv.r

>QUIT

-
```

A directory listing of the disk will show that two new files exist—namely, EX.LST, which is the list file, and EX.COD, which contains the intermediate code. Now compile the programs BUFFER.PAS, SEEKEX.PAS, and ERROR.PAS.

Note that ERROR.PAS has a bug in it which is present to show error formats and is for demonstration purposes only. It does not affect the running of the program.

The message produced by the error is:

```
ERROR := sum;  { cause an error }
    ^

Line 15, Procedure: ADD        Error: 104
```

The arrow (^) points to the location in the line where the error occurred.

Running the Demonstration Programs

When these programs have been compiled and run, you have checked out your disk and have mastered the fundamentals of the PASCAL-80 facilities.

Addition and Subtraction of Numbers

```
-PASCAL

PASCAL-80 Vv.r

>EX
```

NOTE

If you have not returned to ISIS-II (i.e., QUIT) then you type only:

```
>EX
```

since you are already in the PASCAL-80 System. Also note that the code file produced by compiling EX.PAS is EX.COD, but to run EX.COD, only EX need be typed since the PASCAL-80 System automatically appends .COD to any name typed in. Exceptions to this condition are explained in Chapter 4 under Code Filename Specification.

The program is now running:

```
Input two integers: 2,5

The difference of the two is: -3
The sum of the two is: 7

PASCAL-80 V.vr

>
```

EX.PAS Program Listing

```
Line  Seg  Proc  Lev  Disp

  1    1    1           1   program example;
  2    1    1           3
  3    1    1           3   var i, j:  integer;
  4    1    1           5
  5    1    2           3     function sub (value1, value2:  integer):  integer;
  6    1    2     0     0     begin
  7    1    2     1     0       sub:= value1 - value2;
  8    1    2     0     5     end { sub } ;
  9    1    2     0    18
 10    1    3           1    procedure diff__and__sum (var sum, difference:  integer);
 11    1    3           3     var temp:  integer;
 12    1    3           4
 13    1    4           3       function add (value1, value2:  integer):  integer;
 14    1    4     0     0       begin
 15    1    4     1     0         add:= value1 + value2;
 16    1    4     0     5       end { add } ;
 17    1    4     0    18
 18    1    3     0     0     begin { diff__and__sum }
 19    1    3     1     0       temp:= add(sum, difference);
 20    1    3     1    10       difference:= sub(sum, difference);
 21    1    3     1    20       sum:= temp;
 22    1    3     0    23     end { diff__and__sum } ;
 23    1    3     0    36
 24    1    1     0     0   begin { example }
 25    1    1     1     0     writeln; writeln; writeln;
 26    1    1     1    26     write(‘Input two integers: ’);
 27    1    1     1    57     readln(i, j);
 28    1    1     1    85     if (i <>0) or (j <> 0) then
 29    1    1     2    94       begin
 30    1    1     3    94         diff__and__sum (i, j);
 31    1    1     3   100         writeln;
 32    1    1     2   108       end;
 33    1    1     1   108     writeln;
 34    1    1     1   116     writeln(‘The difference of the two is: ’,j:0);
 35    1    1     1   175     writeln(‘The sum of the two is: ’,i:0);
 36    1    1     1   227     writeln;
 37    1    1     0   235   end { example } .
```

Bufferread and Bufferwrite Example

```
>BUFFER
Input a line of text:
THIS IS A LINE OF TEXT.

 Input break char [cntrl Z to stop]: A
 The buffer read:
 THIS IS A
 Length: 9

 Input break char [cntrl Z to stop]: W
 The buffer read:
 THIS IS A LINE OF TEXT.
 Length: 23
 (Break char not found)

 Input break char [cntrl Z to stop]: <cntrl Z>

 PASCAL-80 Vv.r

 >QUIT

 -
```

BUFFER.PAS Program Listing

```
Line  Seg  Proc  Lev  Disp

  1    1    1           1   program example;
  2    1    1           3
  3    1    1           3   { Example using bufferread and bufferwrite with break characters }
  4    1    1           3
  5    1    1           3   var buffer: string;
  6    1    1          44       disk_storage: file;
  7    1    1          64       break: char;
  8    1    1          65       new_len, len: integer;
  9    1    1          67       buff_array: packed array[0..80] of char;
 10    1    1         108
 11    1    1     0     0   begin
 12    1    1     1     0     rewrite (disk_storage, 'data');
 13    1    1     1    27     writeln('Input a line of text: ');
 14    1    1     1    68     readln (buffer);
 15    1    1     1    87     len := bufferwrite(disk_storage, buffer[1], length(buffer));
 16    1    1     1   109     repeat
 17    1    1     2   109       reset(disk_storage);
 18    1    1     2   116       writeln; writeln;
 19    1    1     2   132       write('Input break char [cntrl Z to stop]: ');
 20    1    1     2   179       readln(break);
 21    1    1     2   197       if not eof(input) then
 22    1    1     3   208         begin
 23    1    1     4   208           new_len := bufferread(disk_storage, buff_array, len, ord(break));
 24    1    1     4   226           writeln('The buffer read: ');
 25    1    1     4   262           writeln(copy(buffer, 1, abs(new_len)));
 26    1    1     4   292           writeln('Length: ', abs(new_len):0);
 27    1    1     4   331           if new_len < 0 then writeln('(Break char not found)');
 28    1    1     3   378         end;
 29    1    1     1   378     until eof(input);
 30    1    1     0   388   end.
```

ERROR.PAS Program Listing

```
        Line  Seg  Proc  Lev  Disp

          1    1    1          1   program example;
          2    1    1          3
          3    1    1          3   var i,j: integer;
          4    1    1          5
          5    1    2          3     function sub (value1, value2: integer): integer;
          6    1    2     0    0     begin
          7    1    2     1    0       sub:= value1 - value2;
          8    1    2     0    5     end { sub } ;
          9    1    2     0   18
         10    1    3          1     procedure diff__and__sum (var sum, difference: integer);
         11    1    3          3     var temp: integer;
         12    1    3          4
         13    1    4          3       function add (value1, value2: integer): integer;
         14    1    4     0    0       begin
         15    1    4     0    0
         16    1    4     1    0   ERROR  := sum;  { cause an error }
ERROR 104
         17    1    4     1    4
         18    1    4     1    4         add:= value1 + value2;
         19    1    4     0    9       end { add } ;
         20    1    4     0   22
         21    1    3     0    0     begin { diff__and__sum }
         22    1    3     1    0       temp:= add(sum, difference);
         23    1    3     1   10       difference:= sub(sum, difference);
         24    1    3     1   20       sum:= temp;
         25    1    3     0   23     end { diff__and__sum } ;
         26    1    3     0   36
         27    1    1     0    0   begin { example }
         28    1    1     1    0     writeln; writeln; writeln;
         29    1    1     1   26     write('Input two integers: ');
         30    1    1     1   57     readln(i, j);
         31    1    1     1   85     if (i <> 0) or (j <> 0) then
         32    1    1     2   94       begin
         33    1    1     3   94         diff__and__sum (i, j);
         34         1     3  100         writeln;
         35    1    1     2  108       end;
         36    1    1     1  108     writeln;
         37    1    1     1  116     writeln('The difference of the two is: ',j:0);
         38    1    1     1  175     writeln('The sum of the two is: ',i:0);
         39    1    1     1  227     writeln;
         40    1    1     0  235   end { example } .
```

Example of Seek and Advanced Record Structure

This program is an information retrieval program named SEEKEX.PAS, which is designed to manage "user information records" in a data file named PEOPLE.DAT. Each record in this file contains the name of a person, along with additional personal information. SEEKEX will display a requested record and will allow the user to update the data in the record.

Information Retrieval Program: SEEKEX.PAS

```
-PASCAL SEEKEX
```

This starts SEEKEX.PAS running, assuming it has been compiled. If it has not been compiled yet, do so by typing:

```
-PASCAL COMP SEEKEX.PAS
```

and then run it by typing:

```
>SEEKEX
```

The system will respond with the following:

```
Display which record [Enter number] : 10

    NO SUCH RECORD

Display which record [Enter number] : 5

    Criss Mathews
    Female
    Social security number : 435
    Birth date : 5/30/75
    Number of dependents : 0
    Debts : 0
    Single

Do you wish to change this record? Y

    First name: XXXX
    Last name: YYYY
    S.S. number: 999
    M(ale) or F(emale): M

Input birth date... Month: 5
                    Day: 16
                    Year: 58

    Number of dependents: 0
    Amount of money owed: 0
    Marital Status—[M(arried), S(ingle), D(ivorced), W(idowed)]: S

Display which record [Enter number] : <cntrl Z>

    PASCAL-80 Vv.r

    >QUIT

    -
```

The record numbers that exist are numbered 0 through 8.

SEEKEX.PAS Program Listing

```
program demo_seek;
type
  status = (married, widowed, divorced, single);
  date = record
           month: (jan, feb, mar, apr, may, jun,
                   jul, aug, sep, oct, nov, dec);
           day: 1..31;
           year: integer;
         end;
  person = record
             allocated: boolean;
             name: record
                     first, last: string[20];
                   end;
             ssnum: integer;
             sex: (male, female);
             birth: date;
             dependents: integer;
             debts: integer;
             case marital_status : status of
               married, widowed: (date_married: date);
               divorced: (date_divorced: date; first_divorce: boolean);
               single: (independent: boolean);
           end { person };

var people: file of person;
    ch: char;
    recnumber: integer;
procedure initfile (only_one: boolean);

var i: integer;
    ch: char;
  procedure get_date(var date_to_set : date);
  begin
    with date_to_set do
      begin
        repeat
          readln(i);
          case i of
            1: month := jan;    2: month := feb;    3: month := mar;
            4: month := apr;    5: month := may;    6: month := jun;
            7: month := jul;    8: month := aug;    9: month := sep;
           10: month := oct;   11: month := nov;   12: month := dec;
          end {case};
          if (i< 1) or (i> 12) then write('Bad date.. Try again: ');
        until i in [1..12];
        write('                    DAY: ');
        repeat
          readln(i);
          if (i< 1) or (i> 31) then write('Bad date.. Try again: ');
        until i in [1..31];
        day := i;
        write('                    YEAR: ');
        readln(year);
      end {with};
  end {get_date};

begin {init_files}
  if not only_one then rewrite(people, 'people.dat')
  else seek(people, recnumber);
  repeat
    with people^ do
      begin
        writeln;
        write('First name: ');
        readln(input, name.first);
        if not eof(input) then
          begin
            write('Last name: ');
            readln(input, name.last);
            write('S.S. number: ');
            readln(input,ssnum);
            repeat
              write('M(ale) or F(emale): ');
              readln(input, ch);
              case ch of
                'M','m' : sex := male;
                'F','f' : sex := female;
              end;
```

SEEKEX.PAS Program Listing (Cont'd.)

```
            until ch in ['M','m','F','f'];
            writeln;
            write('Input birth date... MONTH: ');
            get_date (birth);
            writeln;
            write('Number of dependents: ');
            readln(input, dependents);
            write('Amount of money owed: ');
            readln(input, debts);
            repeat
              write('Marital Status-- [M(arried), S(ingle), ');
              write('D(ivorced), W(idowed)]: ');
              readln(input, ch);
              case ch of
                'M','m': begin
                           marital_status := married;
                           write('Input date married .. MONTH: ');
                           get_date(date_married);
                         end;
                'S','s': begin
                           marital_status := single;
                           independent := true;
                         end;
                'D','d': begin
                           marital_status := divorced;
                           first_divorse := true;
                           write('Input date divorced .. MONTH: ');
                           get_date(date_divorsed);
                         end;
                'W','w': begin
                           marital_status := widowed;
                           write('Input date married .. MONTH: ');
                           get_date(date_married);
                         end;
              end {case};
            until ch in ['M','m','S','s','D','d','W','w'];
            allocated := true;
            put(people);
            writeln;
          end;
        end {with};
  until eof(input) or only_one;
  if not only_one then close(people, lock);
end {initfile};

function display (a_record: integer) : boolean;

  procedure print_date (date_to_print: date);
  begin
    with date_to_print do
      begin
        writeln(ord(month)+1:0,'/',day:0,'/',year:0);
      end {with};
  end {print_date};

begin {display}
  seek(people, a_record);
  get(people);
  with people^, name do
    begin
      if eof(people) or not allocated then
        begin
          display := false;
          exit(display);


        end;
      display := true;
      writeln;
      writeln;
      writeln(first,' ',last);
      case sex of
        male: writeln('Male');
        female: writeln('Female');
      end {case};
      writeln('Social security number : ', ssnum:0);
      write('Birth date : ');
      print date(birth);
```

SEEKEX.PAS Program Listing (Cont'd.)

```
      writeln('Number of dependents : ', dependents:0);
      writeln('Debts : ', debts:0);
      case marital_status of
        married:   begin
                     write('Married on : ');
                     print_date(date_married);
                   end;
        widowed:   begin
                     write('Widowed, married on : ');
                     print_date(date_married);
                   end;
        divorced:  begin
                     write('Divorced on : ');
                     print_date(date_divorced);
                   end;
        single:    writeln('Single');
      end {case};
    end {with};
  writeln;
  writeln;
end {display};


begin
  reset(people,'people.dat');
  repeat
    writeln;
    write('Display which record [Enter number] : ');
    readln(recnumber);
    if not eof(input) then
      if display(recnumber) then
        begin
          write('Do you wish to change this record? ');
          readln(ch);
          if ch in ['y','Y'] then initfile (true);
        end
      else writeln('NO SUCH RECORD');
  until eof(input);
end.
```

CHAPTER 6 SEPARATE COMPILATION, PROGRAM LINKAGE, RELOCATION, AND EXECUTION

The previous chapters describe the procedures for compiling and executing a PASCAL-80 program consisting of a single compilation unit. This chapter describes the techniques available for partitioning a large program into separate compilation units, for linking a PASCAL-80 program with external modules written in PL/M-80, FORTRAN-80, or ASM-80, and for constructing a load-and-go version of your Pascal program which is directly executable from the ISIS-II command line interpreter. All of Chapter 6 exceeds standard Pascal.

Partitioning a Pascal Program

In order to allow for easier development of large PASCAL-80 programs, a program may be partitioned into separate **COMPONENTs** or compilation units. These COMPONENTs are then compiled separately and combined by the JOIN utility to produce the final code file.

Since the COMPONENTs are compiled separately, a change in a COMPONENT requires that only that COMPONENT be recompiled, not the entire program. This recompilation is then followed by the JOIN procedure. The entire program must be recompiled only if the global data in the main program is altered.

The Structure of a Partitioned Program

A **partitioned program set** consists of a main **PARTITIONED** program and one or more separate COMPONENTs. The main PARTITIONED program contains the global data and the outer block program statements, as well as all resident global procedures. Each COMPONENT contains one or more global segment procedures. Each one of the COMPONENTs, as well as the main PARTITIONED program, is a compilation unit and is compiled separately.

Each global segment procedure which is contained in a COMPONENT must be declared in the main program as a **SEPARATE** segment procedure, as follows:

```
segment procedure pl;
  separate;
```

Note that the syntax of a SEPARATE segment procedure declaration is similar to that of a forward segment procedure declaration.

The syntax of a PARTITIONED program is as follows:

```
<partitioned program> ::= <partitioned program heading> <block> .
<partitioned program heading> ::= PARTITIONED PROGRAM <ident> |
            PARTITIONED PROGRAM <ident>(<file id> {,<file id>});

<block> ::= <label declaration part>
            <constant declaration part>
            <type declaration part>
            <variable declaration part>
            <separate procedure and function heading part>
            <procedure and function heading part>
            <statement part>
```

```
<separate procedure and function heading part> ::=
              {<separate procedure or function heading> ;}

<separate procedure or function heading> ::=
              <separate procedure heading> | <separate function heading>

<separate procedure heading> ::= SEGMENT <procedure heading> SEPARATE;

<separate function heading> ::= SEGMENT <function heading> SEPARATE;
```

The syntax of a COMPONENT is as follows:

```
<component> ::= <component heading> <component block> .

<component heading> ::= COMPONENT <identifier>;

<component block> ::= <segement procedure and function declaration part>
                      BEGIN END

<segment procedure and function declaration part> ::=
                      {<segment procedure or function declaration> ;}

<segment procedure or function declaration> ::=
          <segment procedure declaration> | <segment function declaration>

<segment procedure declaration> ::= SEGMENT <procedure declaration>

<segment function declaraction> ::= SEGMENT <function declaration>
```

For an example of a Pascal program which has been divided into three compilation units, figure 6-1 should be referenced. As is seen in this example, three SEGMENT PROCEDUREs have been removed from the main program and placed in the COMPONENTs. The complete program is composed of a main PARTITIONED PROGRAM, six1, and two COMPONENTs, six1c1 and six1c2. Each of these compilation units resides on a separate diskette file: The PARTITIONED PROGRAM six1 is on SIX1.PAM, the COMPONENT six1c1 is on SIX1.PA1, and the COMPONENT six1c2 is on SIX1.PA2.

Note that the structure of both a PARTITIONED PROGRAM and a COMPONENT are similar to that of a (standard) PROGRAM. The differences are noted below.

A PARTITIONED PROGRAM has the following characteristics:

- The reserved word PARTITIONED in the program heading.
- The presence of one or more SEPARATE PROCEDURE or FUNCTION declarations.

A COMPONENT has the following characteristics:

- The reserved word COMPONENT which replaces PROGRAM in the heading.
- No global labels, constants, types or variables may be declared; the complete global environment is declared in the main PARTITIONED PROGRAM.
- There are no program statements in the outer block; just a BEGIN followed by an END.
- Each SEPARATE PROCEDURE and FUNCTION must be declared as a SEGMENT on the global level.
- Each SEPARATE PROCEDURE and FUNCTION can reference every global constant, type, variable, PROCEDURE and FUNCTION.

```
{ Main Program Compilation Unit - Source on SIX1.PAM }

partitioned program six1;
  const header = 'This is procedure ';
  type procname = string[10];
  var level: integer;
  segment procedure aa;
    separate;
  segment procedure bb;
    separate;
  segment procedure cc;
    separate;
  function increment(i: integer): integer;
    begin
      writeln('    level: ',i:3);
      increment := i + 1;
    end; { increment }

  begin { six1 }
    level := 1;
    write('This is the main program');
    level := increment(level);
    aa;
  end. { six1 }

{ Component 1 Compilation Unit - Source on SIX1.PA1 }

component six1c1;
  segment procedure aa;
    var name: procname;
    begin
      name := 'aa';
      write(header,name);
      level := increment(level);
      bb;
    end; { aa }

  begin
  end. { six1c1 }

{ Component 2 Compilation Unit - Source on SIX1.PA2 }

component six1c2;
  segment procedure bb;
    begin
      write(header,'bb');
      level := increment(level);
      cc;
    end; { bb }
  segment procedure cc;
    begin
      write(header,'cc');
      level := increment(level);
    end; { cc }

  begin
  end. { six1c2 }
```

Figure 6-1. A Partitioned Pascal Program

Code File Construction

The final, executable, code file is constructed from the code files of all of the compilation units. The main PARTITIONED PROGRAM and each of the COMPONENTs are compiled separately. The PARTITIONED PROGRAM must be compiled before any of the COMPONENTs can be compiled.

During the compilation of the PARTITIONED PROGRAM, the compiler writes the global symbol table onto a diskette file. This global symbol table file is then read in during the compilation of each of the COMPONENTs. In this way, each of the SEPARATE PROCEDUREs and FUNCTIONs of a partitioned program set is declared within the global environment declared in the main PARTITIONED PROGRAM. The GLOBAL(<file-name>) compiler directive allows the name of the global symbol table file to be specified. In the absence of this directive, the compiler uses the file name:

    <source-file-name>.SYM

For example, if SIX1.PAM is a PARTITIONED PROGRAM, the command:

```
COMP SIX1.PAM GLOBAL(MYSYM.XYZ)
```

will cause the global symbol table to be written onto MYSYM.XYZ, while the command:

```
COMP SIX1.PAM
```

will cause the global symbol table to be placed on the file SIX1.SYM.

After all of the compilation units of a partitioned program set have been successfully compiled, the JOIN utility is used to combine all of the separate code files into a final, executable code file. The format of the JOIN command is:

```
JOIN <code-file> {,<code-file>} TO <code-file>
```

For example, the following command will combine SIX1.COM, SIX1.CO1 and SIX1.CO2 into SIX1.COD:

```
JOIN SIX1.COM,SIX1.CO1,SIX1.CO2 TO SIX1.COD
```

The sequence of commands shown in figure 6-2 will compile each of the compilation units shown in figure 6-1 and then combine the resultant code files into an executable code file. The file SIX1.SYM is used as the global symbol table file, since none of the compilation commands specify a GLOBAL directive. As an aid to understanding this facility, the release diskette contains the three source files: SIX1.PAM, SIX1.PA1 and SIX1.PA2, as well as the file SIX1.CSD, which is a SUBMIT file containing the commands shown in figure 6-2.

Linking with non-Pascal Modules

For some applications, it may be necessary or desirable to code one or more procedures or functions in a language other than Pascal. These external procedures and functions can be written in PL/M-80, FORTRAN-80, or 8080/8085 Assembly Language and translated by the appropriate language processor. The resultant relocatable object module(s) can be linked, located and then executed as an integral part of the PASCAL-80 program. Refer to other Intel manuals, cited in the preface, for instructions on translating and linking subroutines written in other languages.

In the remainder of this section, a set of external procedures and functions, together with any associated data structures will be referred to as an **external module**.

PASCAL-80 Extensions

In order to be able to communicate with an external module, a Pascal program must be able to specify what variables, procedures and/or functions are, in fact, external to the compilation.

```
PASCAL
COMP :F1:SIX1.PAM CODE(:F1:SIX1.COM)
COMP :F1:SIX1.PA1 CODE(:F1:SIX1.CO1)
COMP :F1:SIX1.PA2 CODE(:F1:SIX1.CO2)
JOIN :F1:SIX1.COM,:F1:SIX1.CO1,:F1:SIX1.CO2 TO :F1:SIX1.COD
QUIT
```

Figure 6-2. Compiling and Joining a Partitioned Pascal Program

These specifications serve two purposes. First, they declare the objects so that the Pascal program may manipulate them as if they were normal (i.e., non-external) objects. Second, the specifications cause external references to be generated, so that the LINK utility may combine the external modules with the Pascal program.

A PASCAL-80 program declares external variables, procedures and functions in a **Public Declaration Record**. The syntax of a public declaration record is as follows:

```
<public declaration record> ::=
      PUBLIC [<name>];
         <constant declaration part>
         <type declaration part>
         <variable declaration part>
         <procedure and function heading part>
      END;
```

An example of a public declaration record is:

```
public asum;
   type range = 0..1000;
   var aresult: integer;
   function asum(a,b,c: range): integer;
end;
```

All of the items defined in a public declaration record are defined in the global environment of the Pascal program. The only difference between a normal declaration and declaration within a public declaration record is that the former causes storage to be allocated within the Pascal program, while the latter causes the Pascal program to access storage locations allocated externally to the program. In all other ways, a PASCAL-80 program treats private variables and external variables identically.

NOTE

It is the programmer's responsibility to ensure that the structure and size of an external variable is consistent with the declaration of the variable within the public declaration record. If this is not so, unpredictable and usually disastrous results will occur.

The <name> field is an optional field. If it is present, it is accepted by the compiler but otherwise ignored.

Since the <constant declaration part> and <type declaration part> do not cause any storage to be allocated, they are not explicitly associated with any external storage locations. However, they are necessary within a public declaration record so that the variables, procedures, and functions may be defined adequately.

The <variable declaration part> declares variables whose storage is allocated externally to the Pascal program. The compiler allocates an **Indirect Reference Pointer** in the global environment for each such external variable. An indirect reference pointer is one word in length and is used to address the external variable.

The <procedure and function heading part> declares procedures and functions which are defined and allocated externally to the Pascal program. The compiler allocates an indirect reference pointer in the global environment for each such external procedure or function. These procedures are assumed to exist in memory in 8080/8085 machine-executable form and the Pascal interpreter CALLs them using the standard PL/M subroutine calling convention. Since the only two data types which can be passed as parameters using this convention are the BYTE type and the ADDRESS type, a procedure or function heading within a public declaration record has a restricted form. In particular, if the size of a parameter is not greater than one word (two bytes) it may be passed by value or by reference (VAR). However, if the size of a parameter is greater than one word, it must be a VAR parameter. Additionally, the size of the result of a function must not be greater than one word.

The general structure of a program which accesses external objects is as follows:

```
<program> ::= <program heading> <block>

<block> ::= <public declaration part>
            <label declaration part>
            <constant declaration part>
            <type declaration part>
            <variable declaration part>
            <procedure and function declaration part>
            <statement part>

<public declaration part> ::=
                <empty> | { <public declaration record> }
```

An example of such a program is shown in figure 6-3. This Pascal program references two external modules, ASUM and PSUM, each of which declare both an integer variable and an integer function with three parameters of type range. The module ASUM is written in 8080/8085 Assembly Language and the module PSUM is written in PL/M-80. Each of the external functions merely computes the sum of their three parameters, stores the sum in a PUBLIC variable, and then returns the result as their function value. As is shown in this example, the module coded in PL/M must not be a main module; i.e., it must not contain executable statements at the module level.

Note that there may be multiple public declaration records, but that they must all appear before any private objects are declared. In particular, this restriction causes all external objects to be in the global environment of the Pascal program. Therefore, as for all global identifiers, every external variable, procedure, and function name must be different from every other external name, and from every private global variable name. If this condition is not satisfied (e.g., if two external names are the same, or if an external name is the same as a private global name), the compiler will generate Error 101: Identifier declared twice, and will not accept the duplicate name.

However, it is possible that two separate public declaration records may contain declarations for variables of the same type. If this type is not a predeclared data type, these public declaration records will duplicate type declarations. In order to allow for this situation, and not cause a duplicate type or constant declaration error, the compiler will allow types and constants to be redeclared in public declaration records if the new declaration is identical to the original.

Figure 6-3 contains an example of a type, range, which is declared in one public declaration record and then redeclared in another. This situation is necessary since each module must declare the range type independently of any other module.

A public declaration record may be *included* in the source file, using the {$I<filename>} construct, in the same way as any other Pascal source text, as is demonstrated in figure 6-3 (see Appendix C for details).

```
{ Program with external linkages - Source on SIX3.PAS }

program six3;

  public asum;
    type range = 0..1000;
    var arslt: integer;
    function asum(a,b,c: range): integer;
  end; { asum }

  public psum;
    type range = 0..1000;
    var prslt: integer;
    function psum(a,b,c: range): integer;
  end;

  var i,j,k: range;
      result: integer;
  begin
    i := 10; j := 12; k := 14;
    result := asum(i,j,k);
    writeln(result,arslt);
    result := psum(i,j,k);
    writeln(result,prslt);
  end. { six3 }



;
; External module to compute the SUM of the three arguments:
;
;       function asum(a,b,c: 0..1000): integer
;
;
        NAME      ASUM

        PUBLIC    ARSLT,ASUM

; Declare the result variable

        DSEG
ARSLT:  DW        0

        CSEG

; Compute the sum of the three arguments

ASUM:   XCHG                ; Move DE to HL
        DAD       B         ; Add in the second argument
        POP       D         ; Skip over the return address
        POP       B         ;  get the third argument
        DAD       B         ;  and add it in
        PUSH      D         ; Restore return address
        SHLD      ARSLT
        RET

        END                 ; No starting address, since not a main module


/*  External module to compute the SUM of the three arguments: */

/*        function psum(a,b,c: 0..1000): integer               */

PSUM$MODULE:
  DO;

    DECLARE PRSLT ADDRESS PUBLIC;

    PSUM: PROCEDURE(A,B,C) ADDRESS PUBLIC;
      DECLARE(A,B,C) ADDRESS;
      PRSLT = A + B + C;
      RETURN PRSLT;
    END PSUM;

  END PSUM$MODULE;
```

Figure 6-3. A Pascal Program With External Linkages

Invoking an External Procedure or Function

As was discussed above, except for the physical location of the object, external objects and private objects are treated identically by the compiler at the source language level. This implies, in particular, that a Pascal program invokes an external procedure or function in exactly the same manner as a private Pascal procedure or function. The necessary linkages to the external routine are handled by the compiler and the interpreter.

From the point of view of the external module, an external procedure or function is called from a Pascal program by the Pascal interpreter using the standard PL/M calling convention. The details of that convention and their relationship to PASCAL-80, are as follows:

- The value of an actual parameter is passed to an external procedure if the corresponding formal parameter is specified as a value parameter. The address of the actual parameter is passed if the corresponding formal parameter is specified as a VAR, or variable, parameter.
- If the external procedure for function is coded in PL/M or FORTRAN, each (Pascal) actual parameter is assigned to its corresponding (PL/M or FORTRAN) formal parameter.
- If the external procedure or function is coded in assembly language, it will find its arguments in the standard locations:
  - A single parameter is passed in the BC register pair.
  - For a two-parameter procedure or function, the first parameter is passed as above and the second is passed in a similar fashion in the DE register pair.
  - For a procedure or function with more than two parameters, the last two parameters are passed as described above (next to last in BC, last in DE) and the remainder are found on the stack. They are pushed onto the stack in order from left to right in the parameter list, followed by the return location into Pascal, which is the state of the stack when the called procedure or function begins execution.
- The only restriction to the standard PL/M calling convention is that an external PL/M function must be an ADDRESS procedure, even if the value being returned occupies one byte or less. Similarly, if an external function is coded in assembly language, it must always return its value in the HL register pair. If it normally would return a byte in the A register, it should copy the result in A to the L register and clear the H register prior to returning to the Pascal program.

These details of external module linkage are illustrated by the programs in figure 6-3.

The FORTRAN-80 convention for calling a *function* with n arguments is to pass n+1 addresses to the function routine. The first address is the location for storing the result and the next addresses are the locations of the n arguments (since, in FORTRAN-80, all arguments are passed by reference). Therefore, if you want to call an external function of n arguments written in FORTRAN-80 from your Pascal program, you should declare an external *procedure* of n+1 parameters, each of which is a VAR parameter, and the first one is the variable which receives the result of the FORTRAN-80 function.

The External Reference Table

If a PASCAL-80 program references one or more external objects, the compiler automatically generates an **External Reference Table** on an ISIS-II relocatable object file. The ETAB(<file-name>) compiler directive allows the name of the external reference table file to be specified. In the absence of this directive, the compiler uses the file name:

```
<source-file-name>.ERT
```

The External Reference Table contains an external reference for each external object declared in the Pascal program, and defines the relationship between the location of each external object and the location of its Indirect Reference Pointer in the global environment of the Pascal program. The address of the External Reference Table itself is defined by the PUBLIC symbol **PQETAB**.

Program Execution with External Objects

In order to execute a PASCAL-80 program which references external objects, it is necessary to:

1. Compile the Pascal program.
2. Compile and/or assemble the external module(s).
3. Link the external module(s) together with the External Reference Table.
4. Locate the module produced by step 3.
5. Load the module into memory and invoke the PASCAL-80 Run Time System.

Each of these steps is explained in detail below, using the programs in figure 6-3 as an example.

Compile the Pascal Program

If the Pascal program is a single compilation unit, then compile it to produce the code and External Reference Table file. If the Pascal program is a partitioned program, compile the main module and each of the COMPONENTs. The compiler will generate the External Reference Table file when compiling the main module, then invoke the JOIN utility to combine all the code files into one resultant code file.

For example, the command:

```
PASCAL COMP SIX3
```

will compile SIX3.PAS and generate the following three files:

```
SIX3.COD: The executable code file
SIX3.LST: The compiler listing file
SIX3.ERT: The external reference table file
```

Compile and/or Assemble the External Module(s)

Translate each of the external modules into 8080/8085 machine-executable code using an appropriate language processor. Ensure that each external reference in the Pascal program is matched by exactly one PUBLIC declaration in the collection of external modules.

For example, the commands:

```
PLM80 PSUM.PLM
ASM80 ASUM.ASM
```

will produce the following files:

```
PSUM.OBJ: The relocatable object file
PSUM.LST: The compiler listing file
ASUM.OBJ: The relocatable object file
ASUM.LST: The assembler listing file
```

Link the Modules Together

Using the ISIS-II LINK utility, link all of the external object modules together with the External Reference Table and the library file P80EXT.LIB into one relocatable object module.

NOTE

The file containing the External Reference Table **must** be the first file in the list of input files to LINK, so that the External Reference Table itself is situated at the beginning of the external module.

The format of the LINK command is as follows:

```
LINK <ert-file>,P80EXT.LIB,<E1>,..,<Ek> TO <code-file>
```

where <E1>,..<Ek> are the relocatable object files of the external module(s) and any required support library files.

Using our example programs, the appropriate command is:

```
LINK SIX3.ERT,P80EXT.LIB,PSUM.OBJ,ASUM.OBJ,PLM80.LIB TO EXTMOD.OBJ
```

which will create the complete external object module on EXTMOD.OBJ. Since SIX3.ERT is the first file mentioned in the above command, the external reference table will be correctly situated at the beginning of the CODE segment.

Locate the External Module

As is discussed in Chapter 4, during normal operation of PASCAL-80, the Run Time System occupies all of the memory of the Intellec System between the top of ISIS-II and the base of the monitor. However, if a Pascal program references an external module, the code for this module must reside in system memory during the execution of the Pascal program. In order to allow for this situation, the PASCAL-80 Run Time System has a mechanism whereby it can lower the upper boundary of its work space to make room for an external module. The activation of this mechanism is described in the next section.

Use of this mechanism requires that any such external module be located as high as possible in memory, so as not to unduly restrict PASCAL-80's work space. The Run Time System can only use the memory up to the bottom of the external module; any memory space between the top of the external module and the Monitor will be unused by PASCAL-80. The following discussion presents the considerations when locating an external module in a 64K Intellec System.

A 64K Intellec System has a highest usable memory address of F6C0H. Therefore, the base of the external module should be located at:

```
F6C0H - <total length of module>
```

For example, if the length of the external module is 4096 (1000H) bytes, the base of the module must be set to E6C0H or lower in the LOCATE step.

As was mentioned above, the external reference table must be situated at the beginning of the external module. If the linking operation described in the previous section was performed correctly, the external reference table will be situated at the beginning of the CODE segment in the relocatable object file of the external module. Therefore, if the CODE segment is the first segment in the executable object file generated by LOCATE, the external reference table will be positioned correctly. The ISIS-II LOCATE utility will position the CODE segment as the first segment if the ORDER control is not used in the LOCATE command or, if it is used, CODE is mentioned ahead of any other segment name. Use of the LOCATE facility is fully described in the *ISIS-II User's Guide*.

For example:

```
LOCATE EXTMOD.OBJ CODE (0E6C0H)
```

locates the CODE segment of the module (and therefore the entire module) at E6C0H, and writes the absolute object module onto EXTMOD. Unless a private STACK is required, the STACKSIZE(0) control should be specified since the Pascal Run Time System allocates the STACK from its workspace.

As a check on the relative location of the External Reference Table, the PUBLICS option should be invoked in the LOCATE command. This option will cause the value of the symbol PQETAB (among others) to be listed. This value should be exactly the same as the base address specified in the locate command.

Load the External Module into Memory and Invoke the PASCAL-80 Run Time System

The located external module, EXTMOD, can now be loaded into Intellec Memory at its proper address with the DEBUG command:

```
-DEBUG EXTMOD
#nnnn
```

The DEBUG command loads the file into memory and invokes the Monitor. The Monitor responds with the starting address of the module, which should be ignored. The monitor then displays a period (.) and waits for a command to be entered. You should respond with G8 followed by a carriage return, to return to ISIS-II.

At this point, so as not to disturb the external module loaded into the top of memory, you must immediately invoke the PASCAL-80 Run Time System with the command:

```
PASCAL [-R] *
```

The * control causes PASCAL-80 to set the upper boundary of its work space to the value of the PUBLIC symbol PQETAB, which is defined to be the base of the external module.

The PASCAL-80 Run Time System is now used in the standard way to execute the Pascal program. All external references in the program will be automatically linked to the associated external object.

For example, if the command:

```
SIX3
```

is now given to the PASCAL-80 Command Line Interpreter, the example program in figure 6-3 will be executed and will access the specified external functions and variables.

Load and Go Program Generation

In all the discussions up to now, the only way to execute a PASCAL-80 program was to first invoke the Pascal Run Time System, and then the desired program. While this two-step process is valuable during program development, it may be a bit cumbersome for a fully-debugged, production program. To allow for this latter situation, it is possible to construct a Load and Go version of a Pascal program. This Load and Go version is invoked directly from the ISIS-II command line interpreter, does not automatically display any sign-on or start-up message, and thus presents a simpler, more standard, interface to the user.

Generating the Relocatable Object Module

The PASCAL-80 compiler produces a code file which is in a format appropriate for execution by the Pascal Run Time System. However, this format is not compatible with the relocatable object file format, so the code file must be transformed into a relocatable object file with the GENOBJ utility. This utility is invoked by the command:

```
GENOBJ <code-file> TO <obj-file> [FREE(<size>)]
```

where the FREE control specifies the initial size of the free space, from which both the HEAP and the Execution STACK are allocated. The HEAP grows upward from the bottom of the free space while the STACK grows downward from the top.

The amount of free space a Pascal program needs is dependent on both the static and dynamic characteristics of the program. For example, the size of the local data frame of a procedure will affect the STACK space used, as will the calling depth of a set of recursive procedures. In order to assist in the determination of the amount of free space a Pascal program requires, the PASCAL-80 Run Time System has the SIZEON command. This feature (described in detail in Chapter 4) causes the Run Time System to continuously monitor the amount of free space used by a program and to keep track of the maximum value. This value can then be displayed with the STATS command. Remember, however, that the value obtained during one execution run might not be valid during another run, if the program has different execution characteristics.

In addition to monitoring the sizes of the STACK and HEAP during one or more execution runs, the following factors should be considered when computing the required free space size:

- The global data frame is allocated on the STACK when the program begins execution.
- The data frame of a procedure or function is allocated on the STACK only when that procedure or function is active; i.e., only when it is part of the dynamic calling chain.

- If a procedure or function is called recursively, each of its instances has its own data frame allocated on the STACK.
- All parameters to procedures, functions and operators are passed on the STACK.
- The code of an active SEGMENT PROCEDURE or FUNCTION is allocated on the STACK.
- All dynamic variables are allocated on the HEAP by the new(v) predeclared procedure.

A consideration of these factors leads to the following conclusions:

- The longer a procedure call chain is, the more STACK space is used.
- The deeper the level of recursion (of a procedure or function), the more STACK space is used.
- Two procedures which are never active concurrently will not extend the STACK concurrently.
- The more complex a Pascal expression is, the more STACK space is used during its evaluation.

Linking the Object Module

The relocatable object module created from the pascal program code file must now be linked to the Pascal Run Time libraries and associated external modules, if any. The required LINK command has the form:

```
LINK <input-list> TO <link-file> [<link controls>]
```

The <input-list> is a list of ISIS-II files of the form:

```
<program-module>, &
[<external-ref-table>,...,<external-module>,] &
P80RUN.LIB, P80RAR.LIB, P80ISS.LIB
```

where:

- <program-module> is the relocatable object file created from the Pascal program code file by GENOBJ, and should be the first file on the list.
- <external-ref-table> is the file containing the External Reference Table. This item is present only if the Pascal program references external objects.
- <external-module> represents one or more object files which contain the code for the external module(s). This item is present only if the Pascal program references external objects.
- The last three files are the Pascal libraries required for a Pascal program to run under ISIS-II.

The *ISIS-II User's Guide* should be consulted for complete operating instructions for the LINK utility.

Locating the Object Module

The relocatable object module produced by the previous linking operation must now be located to absolute locations. This procedure is performed with the ISIS-II LOCATE utility, as follows:

```
LOCATE <link-file> [TO <output-file>] [<controls>]
```

The only restriction on this command is that the CODE segment must be located to an even byte boundary. Other than this, the user is free in the choice of locations for the various segments.

Program Execution

The linked and located Pascal program may now be loaded and executed simply by presenting its name to the ISIS-II command line interpreter. For example:

```
-MYPROG [<options>]
```

will load and execute this Pascal program, and will not generate any type of automatic sign-on message.

CHAPTER 7 PREPARING PROGRAMS TO RUN UNDER RMX/80™

This chapter describes the use of the special facilities provided to support Pascal programs in the RMX/80 run-time environment—specifically, the PASCAL-80 Run-Time Package for RMX/80 Systems (iSBC803). Use of RMX/80 itself is described only to the extent necessary to explain how to interface PASCAL-80 programs with it; for complete instructions refer to the *RMX/80 User's Guide*. All of Chapter 7 exceeds standard Pascal.

Also described in this chapter is the use of the RMX/80 Interactive Configuration utility (ICU80) in configuring a Pascal-based RMX/80 application system. The ICU80 utility automates the configuration, linking, and locating operations described in the *RMX/80 User's Guide*. It generates the configuration object module, the controller addressable memory (CAM) module, and a submit file containing the link and locate commands required to build the RMX/80 application system. For a complete description of this utility refer to the *RMX/80 Interactive Configuration Utility User's Guide*.

A PASCAL-80 program executing in the RMX/80 environment can avail itself of all of the features of an ISIS-II Pascal program, including the external linking capabilities described in Chapter 6. Additionally, RMX/80 PASCAL contains language extensions which facilitate the incorporation of a Pascal program as a task in the RMX/80 environment. These extensions give an RMX/80 PASCAL program the ability to interface with the complete facilities of RMX/80. This capability, together with the use of the RMX/80 Interactive Configuration Utility (ICU80), allows a user to construct an RMX/80 application system entirely in the Pascal programming language, without the need to resort to either PL/M or assembly language for certain modules.

Under RMX/80, PASCAL-80 input and output operations normally use the full or minimal Terminal Handler and the Disk File System, rather than the ISIS-II functions used in the ISIS-II environment. Alternatively, you can omit the Terminal Handler and/or Disk File System and just use direct port input and output.

Program Structure Under RMX/80

Under RMX/80, programs run as a series of **tasks** under the control of the RMX/80 Nucleus. These tasks communicate with each other by sending and receiving **messages** to and from **exchanges**. One of the tasks in an RMX/80 application system may be coded in Pascal; this program may make full use of all of the language features of PASCAL-80. The other user tasks, if any, in addition to any external subroutines called by the Pascal task, may be written in PL/M-80, FORTRAN-80, or ASM-80, depending on the nature of the functions to be performed.

The RMX/80 task written in Pascal is coded identically to one coded to execute under ISIS-II; no adjustments at the source language level need be made for the difference in environments. This characteristic allows many RMX/80 PASCAL tasks to be thoroughly tested in the ISIS-II environment before being executed under RMX/80.

In PASCAL-80, procedure and function parameters can be passed by value (value parameters) or by reference (variable or **var** parameters). Also, as described in Chapter 6, a Pascal program can establish linkages with external procedures, functions, and data structures. These capabilities, together with PASCAL-80 extensions described in later sections, allow a Pascal program complete access to all of the facilities of RMX/80.

Initialization and Termination

Every PASCAL-80 program exists within a surrounding Run Time Environment. This surrounding environment contains the predeclared procedures and functions described in Chapter 3 and elsewhere, a number of predeclared data identifiers, and the code required to perform all necessary global initialization and termination sequences. The nature of these sequences depends upon the execution environment of the Pascal program, but in all cases is sufficient to correctly execute the program. In particular, it is not necessary, within the RMX/80 environment, for tasks separate from the main Pascal task to invoke any special initialization for termination routines; the Run Time Environment within the Pascal task contains all of these routines.

When a Pascal program terminates, executing under ISIS-II, the Run Time Environment causes all open files to be closed and then transfers control to the ISIS-II Command Line Interpreter. The termination logic in the Run Time Environment of a Pascal task under RMX/80 also closes all open files, but then causes the task to suspend itself forever (by waiting for a message at an empty, private exchange). This means that if you want your Pascal task to be active at all times, it should be coded as a **repeat** statement or a **while** statement, such that the exit condition is never satisfied. If some such statement structure is not utilized, and the program executes its last statement, or an exit(<program-name>) statement is executed, the Pascal task will be suspended forever; the only way to revive it is to activate the hardware reset feature, which will cause it to be reinitialized.

Input and Output

The RMX/80 PASCAL run-time input/output support library, P80RMX.LIB, allows you to invoke the standard predeclared Pascal procedures and functions (RESET, REWRITE, CLOSE, PUT, GET, SEEK, READ, WRITE, BUFFERREAD, BUFFERWRITE, BLOCKREAD, BLOCKWRITE, EOF, EOLN, and IORESULT) for input and output to the Terminal Handler and Disk File System (DFS). No sending of request messages to the Terminal Handler or Disk File System is required; this is all done by the routines in P80RMX.LIB.

File Name Format

In ISIS-II, filenames and extensions are defined by the user, but device names are preestablished by the system. A discussion of this can be found in Chapter 2 of the *ISIS-II User's Guide*. Under RMX/80, filenames and extensions are files on the Disk File System are defined by the user, and DFS device names are defined by the configuration module. The device names :CI: and :CO: are predefined to be input from the Terminal Handler and output to the Terminal Handler respectively.

A complete file designator has the same format in Pascal programs running in both the ISIS-II and RMX/80 environments. This common format is as follows:

```
:<device>:<filename>.<extension>
```

where:

- <device> is a two character alphanumeric designation for a device. If the file is a DFS file, <device> should correspond to a value of the DEVICENAME field in a Device Configuration Table Entry (see ICU80 discussion below). If the file refers to the Terminal Handler, either :CI: or :CO: are valid.
- <filename> is a one to six upper-case, non-blank, alphanumeric characters which specify a portion of the name of a DFS file.

- <extension> is zero to three upper-case, non-blank, alphanumeric characters which are appended to <filename> to completely specify a Disk File System file.

The following are valid RMX/80 PASCAL file names:

:CI:	Input from Terminal Handler
:CO:	Output to Terminal Handler
:F1:MYFILE.TXT	DFS disk file on device F1
:HD:DATA36	DFS disk file on device HD

The Predeclared Files Input and Output

In PASCAL-80, **input** is a predeclared file variable of type INTERACTIVE and **output** is a predeclared file variable of type TEXT. In a Pascal program executing under ISIS-II, the initialization logic in the surrounding environment automatically connects **input** to the ISIS-II :CI: file and **output** to the ISIS-II :CO: file. Under RMX/80, these automatic connections are also made, where :CI: corresponds to terminal input and :CO: to terminal output, as discussed above. Therefore, as under ISIS-II, an RMX/80 PASCAL program does not have to open the terminal files **input** and **output**.

DFS Service Operations

In addition to supporting the data storage and transfer operations of Pascal, the Disk File System in RMX/80 also provides a number of Directory Maintenance Services and other service operations. While procedures corresponding to these service operations are not predeclared in the surrounding environment, they may easily be invoked by a Pascal program using the external linkage features of PASCAL-80.

Run-Time Error Handling

As in the ISIS-II environment, an RMX/80 PASCAL task can specify a user error procedure to be invoked upon detection of a run-time error by the PASCAL-80 Run-Time System. In the absence of such a specification, the default system error procedure will be invoked upon a run-time error. This procedure will attempt to output an error message to the Terminal Handler, will then cause the task to exit and, hence, suspend itself forever.

In order for the Pascal task to retain control over its execution, even in the event of a run-time error, it is necessary for the program to specify its own error procedure(s), as described in Chapter 8. The specific actions of an error procedure are dependent upon the nature of the run-time error, the state of the execution environment, and the structure of the Pascal task itself; such procedures must be designed as an integral part of the total RMX/80 PASCAL application system.

PASCAL-80 Extensions for RMX/80

RMX/80 PASCAL incorporates a number of language extensions which greatly facilitate the construction of a Pascal task, including its communication with other tasks and with the RMX/80 Nucleus. These extensions fall into two categories: a new predeclared type, the MESSAGE type, and additional predeclared procedures and functions.

Need for RMX/80 Extensions

There exists a major incompatibility between the format of data structures declared in PASCAL-80 and several RMX/80 data structures. This may be understood by observing the following conditions of data storage allocation in PASCAL-80:

- Space for data items is allocated in units of words, where a word is a two-byte quantity aligned on an even byte boundary.
- If the size of a data item is less than or equal to one word, the space allocated for the item will never cross a word boundary.

In particular, these conditions imply that a data item which is equal in size to one of type **integer** (two bytes) will always be allocated on a word boundary.

However, the RMX/80 Message Format is incompatible with these conditions in that the HOME EXCHANGE field and the RESPONSE EXCHANGE field, as well as any additional data items in the remainder of the message, are aligned on odd-byte boundaries. It is therefore not possible to define a data type in PASCAL-80 which has the same structure as an RMX/80 message. Consequently, in the absence of any specific language extensions an RMX/80 message would be a cumbersome type to manipulate.

The above incompatibility is resolved by the definition of a new class of data types and a set of additional predeclared procedures and functions in the supporting environment.

The MESSAGE Data Type

RMX/80 PASCAL incorporates the MESSAGE type, which is used to define a class of structured data types corresponding to RMX/80 messages. Each Pascal MESSAGE is one byte longer than its associated RMX/80 message. This additional byte is located at the beginning of the Pascal defined MESSAGE and offsets the fields in the RMX/80 message format by one byte. The result of the inclusion of this extra byte is that, from the viewpoint of Pascal, the HOME EXCHANGE field, the RESPONSE EXCHANGE field, and all succeeding fields are located on word boundaries and, therefore, are simple to define and manipulate in a Pascal program. For example, a DFS Delete Request Message type is declared as follows:

```
type deletemsg = message
                     homeexch: ^exchange;
                     respexch: ^exchange;
                     status: integer;
                     fileptr: ^fnameblock;
                 end;
```

The MESSAGE type is similar to the RECORD type, with the following differences. In a MESSAGE type:

- An unnamed six-byte field preceds the first declared user field.
- Certain parameters to some predeclared procedures must be of type message.
- The PACKED prefix is not allowed.

Note that a MESSAGE type declaration does not declare the LENGTH and TYPE fields of the RMX/80 message. These fields of a message are manipulated by the predeclared procedures **rqsetlt** and **rqgetlt** described below.

In order to allow for the extra byte in these MESSAGE types, the RMX/80 predeclared procedures pass to the external world the address of the second byte in the MESSAGE, not the address of the beginning of the Pascal MESSAGE. Conversely, when an RMX/80 predeclared procedure receives a message address from a nucleus operation, it decrements it by one byte before returning it to the Pascal program.

When a Pascal program declares a private variable to be of type MESSAGE, the compiler allocates storage for it, and the item is addressed as described above. In particular, Pascal addresses the first byte of the MESSAGE while RMX/80 addresses the second byte. In order to allow for a MESSAGE variable to be declared externally, and still be manipulated correctly, the indirect reference pointer of such a variable points to the byte preceding the first byte in the external message. This address transformation, which is performed automatically, transforms the external message into a Pascal MESSAGE, and therefore allows it to be handled identically to a private MESSAGE.

It is not necessary for the programmer to make any adjustment for the extra byte in a MESSAGE; the Pascal compiler and run-time system effectively mask out its presence. In particular, the PUBLIC symbol which names an external RMX/80 message should be bound, as in non-Pascal applications, to the first byte of the LINK field which, from the viewpoint of RMX/80, is the beginning of the message.

Predeclared Procedures and Functions for RMX/80

The following procedures and functions are predeclared in the Run Time Environment, and exist specifically for use by Pascal programs which will execute under RMX/80.

MESSAGE Manipulation Procedures

rqsetlt(m,l,t)

Sets the value of the LENGTH field of MESSAGE m to l and the value of the TYPE field of m to t. Both l and t must be integer expressions.

rqgetlt(m,l,t)

Sets the value of l to the contents of the LENGTH field of MESSAGE m and sets the value of t to the contents of the TYPE field of m. Both l and m must be variables of type integer.

RMX/80 Nucleus Operations

In RMX/80 PASCAL, there is a predeclared procedure (or function) corresponding to each RMX/80 Nucleus operation. The names of these Pascal routines are identical to their RMX/80 counterparts, as are the order and type of their parameters.

rqsend(e,m)

Sends the MESSAGE m to the exchange e.

rqwait(mp,e,t)

Waits at exchange e for a time limit of t, which must be an integer expression, and returns the address of a message in mp, which must be a pointer to a MESSAGE.

rqacpt(mp,e)

Returns the address of a message which has been sent to the exchange e. If no message is there, NIL is returned. The address (or NIL) is placed in mp, which must be a pointer to a MESSAGE.

rqctck

Decrements the time counters on all tasks on the Delay list.

rqctsk(s)

Creates a task, given the Static Task Descriptor s.

rqcxch(e)

Initializes the exchange e.

rqdlvl(l)

Disables the interrupt level given by l, which must be an integer expression.

rqdtsk(t)

Deletes the task described by the task descriptor t.

rqdxch(e): integer

Deletes the exchange e.

rqelvl(l)

Enables the interrupt level given by l, which must be an integer expression.

rqendi

Ends the interrupt in a user-supplied interrupt routine.

rqisnd(i)

Sends a message to the interrupt exchange i.

rqresm(t)

Resumes a suspended task described by the task descriptor t.

rqsetp(a,l)

Associates the address, a, of a user-supplied interrupt procedure with an interrupt level. The procedure cannot be a Pascal procedure. Used for iSBC 80/10 applications.

rqsetv(a,l)

Associates the address, a, of a user-supplied interrupt procedure with an interrupt level. The procedure cannot be a Pascal procedure. Used for non-iSBC 80/10 applications.

rqsusp(t)

Suspends the task described by the task descriptor t.

Constructing an RMX/80 System

The construction of an RMX/80 applications system which contains a Pascal task can be partitioned into four major steps:

- Generating the relocatable object modules of any non-Pascal tasks.
- Generating the relocatable object module of the Pascal task.
- Invoking the RMX/80 Interactive Configuration Utility (ICU80) to produce the various configuration object modules and the associated Command Sequence Definition file.
- Applying the SUBMIT command to the Command Sequence Definition file generated in the above step to link and locate the complete RMX/80 application system.

Since a discussion of the first step above is dependent upon the specific application system being constructed, we will assume any such modules have been compiled/assembled and linked. Refer to the Intel manuals cited in the Preface for instructions about utilizing other languages.

Generating the Relocatable Object Module

The procedure for generating the relocatable object module of a Pascal task is essentially identical to that for constructing the relocatable object module of a Load and Go Pascal program. Accordingly, the section in Chapter 6 which discusses Load and Go program generation should be referenced. The only difference between the two procedures is the format of the GENOBJ command. When generating the relocatable object module of an RMX/80 PASCAL task, this utility is invoked by:

```
GENOBJ <code-file> TO <obj-file> [FREE(n)] RMX
```

where the inclusion of the RMX option causes GENOBJ to reference RMX/80-specific Resident System procedures, instead of the default ISIS-II-specific procedures.

The Interactive Configuration Utility

The RMX/80 Interactive Configuration Utility (ICU80) is a utility used to automate the configuration, linking, and locating operations described in the *RMX/80 User's Guide*. ICU80 is used to generate the configuration object module, the controller addressable memory (CAM) module, and a submit file containing the link and locate commands required to build the RMX/80 application system.

ICU80 obtains its input data from two sources: a description file suplied to ICU80 when it is invoked, and data entered from the console by the user in response to prompts from ICU80. One of the features of ICU80 is that an updated description file can be generated which contains all of the additions and modifications the user made during the session with ICU80. This file may then be used as the initial description file the next time ICU80 is invoked.

When using ICU80 to configure an RMX/80 Application System which includes a Pascal task, a number of the configuration parameters must have specific values. These parameters, with their values, are described below. Any configuration parameter which is not mentioned in the following list is not constrained by RMX/80 PASCAL; it may assume any valid value consistent with the structure of the application system.

The values of the following configuration parameters are constrained as follows:

TERM HNDLR:

If the Pascal program references either of the predeclared files input or output or performs I/O to either :CI: or :CO:, one of the versions of the Terminal Handler must be specified.

TASK NAME:

You may use any name for your Pascal task.

ENTRY POINT:

The ENTRY POINT must be specified as P8INIT.

STK LENGTH:

This parameter should be set to the value 2, since the Pascal task defines its own work space and STACK.

DFLT EXCHG:

The Pascal task does not use a Default Exchange.

EXTRA:

This parameter is set to 0.

DFS:

The Disk File System must be included if the Pascal program accesses any disk files.

ATTRIB, DELETE, FORMAT,
LOAD, RENAME, DISKIO:

These DFS services are included only if the Pascal program calls them explicitly.

OPEN, CLOSE,
READ, WRITE, SEEK:

These must be included if the DFS is specified.

LINK:

The following files must be included in the LINK list, in the following order:

```
<program-module>
<ext-ref-table>
<external-modules>
P80RUN.LIB
P80RAR.LIB
P80RMX.LIB
```

where:

- <program-module> is the relocatable object file created from the Pascal program code file by GENOBJ.

- <ext-ref-table> is the file containing the External Reference Table. This item is present only if the Pascal program references external objects.
- <external-modules> represents one or more relocatable object files which contain the code for the external module(s). This item is present only if the Pascal program references external objects.
- The last three files are the PASCAL-80 libraries required for a Pascal program to run in the RMX/80 environment.

CHAPTER 8
RUN-TIME ERROR PROCEDURES

During the execution of a PASCAL-80 program, various run-time errors may occur. These errors are divided into three general categories:

- Data structure errors
- Value range errors
- Input/output errors.

The interpreter will always detect data structure errors, while detection of errors in the other two categories is not automatic. Errors in these categories are detected only by in-line interpreter p-code operations which may or may not be generated by the compiler during the compilation of the Pascal program. As is discussed in Appendix C, the **{$R+/−}** compiler directive controls the generation of the range-checking code, while the **{$I+/−}** directive controls the generation of the I/O checking code. If the code of a compiled Pascal program does not contain this error checking code, the respective run-time errors will never be detected. All of Chapter 8 exceeds standard Pascal.

During program execution, the detection of a run-time error by the PASCAL-80 interpreter causes a parameterless error procedure to be invoked. In the absence of an error procedure specification by the running program, a default error procedure is invoked. This default procedure writes an appropriate error message to the file **output**, and then executes exit (<program-name>), which causes program termination. Appendix E lists the error messages which may be generated by the system default error procedure.

If it is necessary for your program to retain control in the event of a run-time error, a user-declared error procedure may be specified with the predeclared procedure:

errorset(p)

where p is a parameterless procedure declared at the global level of the Pascal program. When a run-time error is detected, the procedure p is invoked at the point of the error. If, after performing its actions, p terminates normally, the interpreter automatically causes an immediate exit from the procedure which caused the error. If p executes an exit(q) operation, the interpreter causes an exit from procedure q, as usual. In either case, the procedure which was executing when the run-time error was detected is terminated, and the STACK is unwound appropriately.

In order for a user error procedure to effectively process the run-time error condition, the following predeclared integer variables are present in the Run-Time Environment:

errnumber	The number of the last run-time error
errsegment	The segment number when the error occurred
errprocedure	The procedure number when the error occurred
errdisplacement	The instruction displacement when the error occurred
errsystem	The ISIS-II error number, if errnumber = 10

When a user-declared error procedure is invoked, the active error procedure specification reverts to the standard default error procedure. Therefore, after a run-time error has occured, erroset(p) must be called again if you still want a user error procedure to be active. See figure 8-1 for an example of a user error procedure.

```
{  This program illustrates the use of the errorset procedure  }

program eightl;

  type value = 0..255;

  procedure rterror;
    begin
      writeln;
      writeln('Detection of Error Number: ',errnumber:0);
      writeln;
    end;

  procedure square(i: value);
    begin
      writeln('Input value: ',i:0);
      i := i * i;
      writeln('Squared result: ',i:0);
      writeln;
    end;

  begin
    errorset(rterror);
    square(10);
    square(100);
    square(200);
  end.  { eightl }
```

Executing this program results in the following output:

```
Input value: 10
Squared result: 100

Input value: 100

Detection of Error Number: 1

Input value: 200

Value range error
Segment:            1
Procedure:          3
Instruction:       54
```

Figure 8-1. Use of the errorset Procedure

APPENDIX A SUMMARY OF OPERATIONS

Operator	Operation	Type of Operand(s)	Result Type
:=	assignment	any type except file types	——
arithmetic:			
+ (unary) − (unary)	identity sign inversion	integer or real	same as operand
+ − *	addition subtraction multiplication	integer or real	integer or real
div / mod	integer division real division modulus	integer integer or real integer	integer real integer
relational:			
= <>	equality inequality	scalar, string, set, or pointer	
< >	less than greater than	scalar or string	boolean
<=	less or equal -or- set inclusion	scalar or string set	
>=	greater or equal -or- set inclusion	scalar or string set	
in	set membership	first operand is any scalar, the second is its set type	
logical:			
not or and	negation disjunction conjunction	boolean	boolean
set:			
+ − *	union set difference intersection	any set type T	T
{ ...} (*...*)	comment	any ASCII characters	

APPENDIX B PASCAL-80 VOCABULARY

The PASCAL-80 language vocabulary consists of basic symbols classified into three categories: letters, digits, and special symbols. The special symbols are operators and delimiters, as follows:

Operators

+	−	*	/	:=	.
,	;	:	'	=	..
<>	<	<=	>=	>	↑
(	)	[	]	{	}

Delimiters {reserved words}

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

In this manual, reserved words are written in upper case letters to emphasize their interpretation as single symbols with a fixed meaning. They may not be used in any context other than that explicit in their definition. In particular, they may not be used as identifiers.

Identifiers

These are names denoting constants, types, variables, procedures, and functions. An identifier must begin with a letter which may be followed by any combination and number of letters and digits. However, in PASCAL-80, only the first eight characters of any identifier are significant.

Matching upper and lower case letters are equivalent in identifiers and reserved words. In particular, the two identifiers: **sample** and **SAMPLE** refer to the same variable. Similarly, **packed** and **PACKED** are equivalent names for the same delimiter.

APPENDIX C SUMMARY OF COMPILER DIRECTIVES

Command Line Directives

A description of the available command line directives follows:

NOLIST

No list file is produced.

NOCODE

No intermediate code file is produced.

ERRLIST

The listing is limited to those lines containing syntax errors.

LIST (external-file-name)

Specifies the file to which the listing is to be directed. The default is: <source-filename>.LST.

CODE (external-file-name)

Specifies the file to which the code is to be directed. The default is: <source-filename>.COD.

NOECHO

Error lines are echoed on the console unless this directive is specifed.

GLOBAL (external-file-name)

Specifies the global symbol table file when separately compiling a partitioned Pascal program. The default is: <source-filename>.SYM.

WORKFILE(<device>)

Specifies which diskette device is to be used for the compiler's workfile, P80WRK.TMP. The default value is the device of the output code file.

DATE(<date>)

Specifies the date to be included in the page heading of the compiler listing. The <date> parameter is any sequence of nine or fewer characters not containing parentheses. The default value is all spaces.

NOSTATISTICS

Specifies that the compiler should not accumulate or list the procedure data frame and parameter sizes.

ETAB(external-file-name)

Specifies the external table file when compiling a Pascal program with external references. The default is: <source-filename>.ERT.

Excluded Combinations

Certain of these directives may not be used in combinations. The following table shows which directives are excluded if the directive in the left-hand column is specified.

Directive	Excluded Directives
NOLIST	LIST ERRLIST
NOCODE	CODE

Embedded Directives

Embedded compiler directives are inserted in the source code and have the following form:

```
{$directive[,directive]} or (*$directive[,directive]*)
```

Many of the directives have the form

```
directive+ or directive-
```

If a + or − is not present, + is assumed.

The use of embedded directives is discussed below.

C This directive causes the text following the C to be placed in the code file. This is useful for putting copyright information and other text into the code file. This directive must appear at the top of the program; otherwise, a compile-time error will occur.

Example:

```
{$C Copyright 1979 Intel Corp.}
```

would cause the copyright message to be written to the code file.

I When this directive is followed by a + or −, the I/O checking is affected as follows:

I+ Causes the compiler to generate code after each statement which performs any I/O, in order to check whether the I/O operation was accomplished successfully. In the case of an unsuccessful I/O operation, a run-time error will occur I+ is the default value.

I− Causes the compiler to generate code with no I/O checking. If an I/O error does occur, a run-time error will not occur.

I When this directive is not followed by a + or −, the directive is interpreted as the INCLUDE FILE directive.

Example:

```
{$I <filename>}
```

will cause the compiler to temporarily suspend its processing of the original source file, and to take subsequent source statements from <filename>. The compiler will continue to process <filename> until it reaches its end-of-file, at which point the compiler will resume processing the original source file. If a drive number is specified, (i.e., :F1:filename), the file will be opened on the specified drive; otherwise, the file is opened on the same drive from which the original source was taken. Since all the characters between the I and the } or *) are taken to be the file name, no other directives can follow the include file directive until that directive is closed by a } or *).

An include file may be specified anywhere in the original source file. The text in the included file merely replaces the INCLUDE directive and becomes part of the source program. The resulting program must, of course, be syntactically and semantically correct, or else a compile-time error will occur. However, in order to allow for convenient structuring of INCLUDEd source code, PASCAL-80 relaxes the rules concerning the ordering of declarations. Under these relaxed rules, if a file is included just after a program's or procedure's last data declaration, (just before the start of the code statements), it may contain, within itself, a complete set of CONST, TYPE, and VAR declarations, as well as PROCEDURE and FUNCTION declarations.

An include file cannot contain an include file directive. If this occurs, a compile-time error is generated.

R This directive affects the RANGE CHECKING of the compiler. If the R is followed by a +, range checking code is generated by the compiler such that if a value range error occurs while running the program, a run-time error occurs. When followed by a −, no range checking code is generated; if a value range error does occur while running the program, a run-time error does not occur. The default value is R+.

O This directive determines whether the compiler operates in overlay mode. If the O+ directive is given, the compiler operates in overlay mode. If the O− directive is given, the compiler doesn't operate in overlay mode. In the O− mode there are about 6300 bytes less memory available for symbol table space, but compile time is decreased since the compiler does not have to read from the disk as often as in the O+ mode. O− is therefore very useful for compiling small and moderate-sized programs. O+ is the default value.

T This directive determines whether the compiler generates tracing instructions, and may appear anywhere in the source code. If the T+ directive is given, tracing instructions are inserted in the generated object code. If the T− directive is given, no tracing instructions are generated. The default value is T+.

H This directive specifies a heading to be printed on the second line of each page of the compiler listing, just below the main title line. All characters in the directive following the H become part of the heading.

For example:

```
{H File Translation Program }
```

will cause the message "File Tanslation Program" to be a heading on each page of the compiler listing.

The next two options deal with list file format:

{$+} : Causes a top of page on the list file.
{$-} : Causes a top of page on the list file if the current position on the page is within 10 lines from the bottom of the page.

APPENDIX D COMPILER ERROR MESSAGES

1: Error in simple type
2: Identifier expected
3: 'PROGRAM' expected
4: ')' expected
5: ':' expected
6: Illegal symbol (maybe missing ';' on the line above)
7: Error in parameter list
8: 'OF' expected
9: '(' expected
10: Error in type
11: '[' expected
12: ']' expected
13: 'END' expected
14: ';' expected
15: Integer expected
16: '=' expected
17: 'BEGIN' expected
18: Error in declaration part
19: Error in <field-list>
20: ',' expected
21: '.' expected

50: Error in constant
51: ':=' expected
52: 'THEN' expected
53: 'UNTIL' expected
54: 'DO' expected
55: 'TO' or 'DOWNTO' expected in for statement
56: 'IF' expected
57: 'FILE' expected
58: Error in <factor> (bad expression)
59: Error in variable

101: Identifier declared twice
102: Low bound exceeds high bound
103: Identifier is not of the appropriate class
104: Undeclared identifier
105: Sign not allowed
106: Number expected
107: Incompatible subrange types
108: File not allowed here
109: Type must not be real
110: <tagfield> type must be scalar or subrange
111: Incompatible with <tagfield> part
112: Index type must not be real
113: Index type must be a scalar or a subrange
114: Base type must not be real
115: Base type must be a scalar or a subrange
116: Error in type of standard procedure parameter
117: Unsatisfied forward reference
118: Forward reference type identifier in variable declaration
119: Respecified parameters for a forward declared procedure
120: Function result type must be scalar, subrange, or pointer
121: File value parameter not allowed

122: A forward declared function's result type can not be respecified
123: Missing result type in function declaration
124: F-format for reals only
125: Error in type of standard procedure parameter
126: Number of parameters does not agree with declaration
127: Illegal parameter substitution
128: Result type does not agree with declaration
129: Type conflict of operands
130: Expression is not of set type
131: Tests on equality allowed only
132: Strict inclusion not allowed
133: File comparison not allowed
134: Illegal type of operand(s)
135: Type of operand must be boolean
136: Set element type must be scalar or subrange
137: Set element types must be compatible
138: Type of variable is not array
139: Index type is not compatible with the declaration
140: Type of variable is not record
141: Type of variable must be file or pointer
142: Illegal parameter solution
143: Illegal type of loop control variable
144: Illegal type of expression
145: Type conflict
146: Assignment of files not allowed
147: Label type incompatible with selecting expression
148: Subrange bounds must be scalar
149: Index type must be integer
150: Assignment to standard function is not allowed
151: Assignment to formal function is not allowed
152: No such field in this record
153: Type error in read
154: Actual parameter must be a variable
155: Control variable cannot be formal or non-local
156: Multidefined case label
157: Too many cases in case statement
158: No such variant in this record
159: Real or string tagfields not allowed
160: Previous declaration was not forward
161: Again forward declared
162: Parameter size must be constant
163: Missing variant in declaration
164: Substitution of standard proc/func not allowed
165: Multidefined label
166: Multideclared label
167: Undeclared label
168: Undefined label
169: Error in base set
170: Value parameter expected
171: Standard file was redeclared
172: Undeclared external file
174: Pascal function or procedure expected
190: Previous declaration was segment declaration
191: Separate declaration valid only on global level
192: Separate procedure must be a segment
193: Comment or heading valid only on global level.

201: Error in real number - digit expected
202: String constant must not exceed source line
203: Integer constant exceeds range

250:	Too many scopes of nested identifiers
251:	Too many nested procedures or functions
252:	Too many forward references of procedure entries
253:	Procedure too long
259:	Expression too complicated
261:	Too many segment declarations
300:	Division by zero
302:	Index expression out of bounds
303:	Value to be assigned is out of bounds
304:	Element expression out of range
398:	Implementation restriction
399:	Implementation restriction
400:	Illegal character in text
401:	Unexpected end of input
402:	Error in writing code file, not enough room
403:	Error in reading include file
404:	Error in writing list file, not enough room
405:	Error in reading globals file
406:	Error in writing globals file
407:	Wrong version of globals file

APPENDIX E
RUN-TIME ERROR MESSAGES

In the absence of a user-specified error procedure, a run-time error will cause the system default error procedure to be invoked. This procedure displays an error message and then forces the current program to terminate. Each run-time error causes a specific message to be displayed, along with the Segment number, Procedure number, and Instruction offset of the offending operation. By referring to the compiler listing of the program in question, the cause of the error may be ascertained.

When a run-time error occurs, the display on the screen will be as follows:

```
<error message>

Segment:      <s>
Procedure:    <p>
Instruction:  <i>
```

where <s>, <p>, and <i> are integers, and <error message> is one of the following:

```
Stack overflow    (4)
Value range error    (1)
Exit from uncalled procedure    (3)
Integer overflow    (5)
String overflow    (13)
Divide by zero    (6)
NIL pointer reference    (7)
System I/O error    (10)
Floating point error    (12)
Unimplemented instruction    (11)
Program interrupted by user (Interrupt 3)    (8)
ISIS-II Error #<e>    where <e> denotes an ISIS-II
                      error number
```

where the number following each message is the number of the error.

APPENDIX F ISIS-II ERROR MESSAGES

This appendix lists the error messages issued by the various ISIS-II commands.

The numbered messages are listed in the first section. The unnumbered messages that are issued by specific commands are listed in subsequent sections.

Numbered ISIS-II Error Messages

By convention, error numbers 1-99 inclusive are reserved for errors that originate in or are detected by the resident routines of ISIS-II; error numbers 100-199 inclusive are reserved for user programs; and numbers 200-255 inclusive are used for errors that may be encountered by nonresident system routines. In the following list an asterisk precedes error numbers that are always fatal. The other errors are generally nonfatal unless they are issued by the kCONSOLE system call. See Table 1 and 2 below.

0	No error detected.
1	Limit of 19 buffers exceeded.
2	AFTN does not specify an open file.
3	Attempt to open more than six files simultaneously.
4	Illegal filename specification.
5	Illegal or unrecognized device specification in pathname.
6	Attempt to write to a file open for input.
7	Operation aborted; insufficient disk space.
8	Attempt to read from a file open for output.
9	No more room in disk directory.
10	Pathnames do not specify the same disk.
11	Cannot rename file; name already in use.
12	Attempt to open a file already open.
13	No such file.
14	Attempt to open for writing or to delete or rename a write-protected file.
15	Attempt to load into ISIS-II area or buffer area.
16	Illegal format record.
17	Attempt to rename/delete a non-disk file.
18	Unrecognized system call.
19	Attempt to seek on a non-disk file.
20	Attempt to seek backward past beginning of file.
21	Attempt to rescan a non-lined file.
22	Illegal ACCESS parameter to OPEN or access mode impossible for file specified.
23	No filename specified for a disk file.
24	Disk error (see below).
25	Incorrect specification of echo file to OPEN.
26	Incorrect SWID parameter in ATTRIB system call.
27	Incorrect MODE parameter in SEEK system call.
28	Null file extension.
29	End of file on console input.
30	Drive not ready.
31	Attempted seek on write-only (output) file.
32	Can't delete an open file.
33	Illegal system call parameter.
34	Bad RETSW argument to LOAD.
35	Attempt to extend a file opened for input by seeking past end-of-file.

201	Unrecognized switch.
202	Unrecognized delimiter character.
203	Invalid command syntax.
204	Premature end-of-file.
206	Illegal disk label.
207	No END statement found in input.
208	Checksum error.
209	Illegal records sequence in object module file.
210	Insufficient memory to complete job.
211	Object module record too long.
212	Bad object module record type.
213	Illegal fixup record specified in object module file.
214	Bad parameter in a SUBMIT file.
215	Argument too long in a SUBMIT invocation.
216	Too many parameters in a SUBMIT invocation.
217	Object module record too short.
218	Illegal object module record format.
219	Phase error in LINK.
220	No end-of-file record in object module file.
221	Segment overflow during Link operation.
222	Unrecognized record in object module file.
223	Fixup record pointer is incorrect.
224	Illegal record sequence in object module file in LINK.
225	Illegal module name specified.
226	Module name exceeds 31 characters.
227	Command syntax requires left parenthesis.
228	Command syntax requires right parenthesis.
229	Unrecognized control specified in command.
230	Duplicate symbol found.
231	File already exists.
232	Unrecognized command.
233	Command syntax requires a "TO" clause.
234	File name illegally duplicated in command.
235	File specified in command is not a library file.
236	More than 249 common segments in input files.
237	Specified common segment not found in object file.
238	Illegal stack content record in object file.
239	No module header in input object file.
240	Program exceeds 64K bytes.

When error number 24 occurs, an additional message is output to the console:

```
STATUS=00nn
D=x T=yyy S=zzz
```

where x represents the drive number, yyy the track address, zzz the sector address, and where nn has the following meanings for floppy disks:

01	Deleted record.
02	Data field CRC error.
03	Invalid address mark.
04	Seek error.
08	Address error.
0A	ID field CRC error.
0E	No address mark.
0F	Incorrect data address mark.
10	Data overrun or data underrun.
20	Attempt to write on Write Protected drive.
40	Drive has indicated a Write error.
80	Operation attempted on drive which is not ready.

For hard disks, nn has the following meanings:

01	ID field miscompare.
03	Data Field CRC error.
04	Seek error.
08	Bad sector address.
0A	ID field CRC error.
0B	Protocol violations.
0C	Bad track address.
0E	No ID address mark or sector not found.
0F	Bad data field address mark.
10	Format error.
20	Attempt to write on Write protected drive.
40	Drive has indicated a Write error.
80	Operation attempted on drive which is not ready.

Table F-1. Nonfatal Error Numbers Returned by System Calls

OPEN	3, 4, 5, 9, 12, 13, 14, 22, 23, 25, 28.
READ	2, 8.
WRITE	2, 6.
SEEK	2, 19, 20, 27, 31, 35.
RESCAN	2, 21.
CLOSE	2.
DELETE	4, 5, 13, 14, 17, 23, 28, 32.
RENAME	4, 5, 10, 11, 13, 17, 23, 28.
ATTRIB	4, 5, 13, 23, 26, 28.
CONSOL	None; all errors are fatal.
WHOCON	None.
ERROR	None.
LOAD	3, 4, 5, 12, 13, 22, 23, 28, 34.
EXIT	None.
SPATH	4, 5, 23, 28.

Table F-2. Fatal Errors Issued by System Calls

OPEN	1, 7, 24, 30, 33.
READ	24, 30, 33.
WRITE	7, 24, 30, 33.
SEEK	7, 24, 30, 33.
RESCAN	33.
CLOSE	33.
DELETE	1, 24, 30, 33.
RENAME	1, 24, 30, 33.
ATTRIB	1, 24, 30, 33.
CONSOL	1, 4, 5, 12, 13, 14, 22, 23, 24, 28, 30, 33.
WHOCON	33.
ERROR	33.
LOAD	1, 15, 16, 24, 30, 33.
SPATH	33.

APPENDIX G SYNTAX SUMMARY

The syntax of Pascal is described below using Backus-Naur Form (BNF) notation. This notation uses meta-symbols which belong to the BNF formalism, and are not symbols of the Pascal language. These meta-symbols are:

::= | { }

In addition, syntactic constructs are denoted by English words enclosed in angular brackets, as in <digit>. These words(s) also define the meaning or nature of the construct. The syntactic construct <empty> denotes the null sequence of symbols.

The first symbol, ::=, is used in BNF productions to mean "is defined to be," as in:

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

The bar symbol (|) is used to denote alternation, as in the above production. In other words, <digit> is defined to be either 0, 1, 2, ..., or 9.

The braces, { and } , denote zero or more repetitions of the symbols enclosed in them. In particular, the production

X ::= { Y }

is a short form of the recursive construct

X ::= <empty> | XY

In the following productions, symbols in upper case denote terminal symbols, not syntactic constructs.

The BNF Description of Pascal

```
<program declaration> ::= <program heading> <block> .

<program heading> ::= PROGRAM <identifier>; |
                      PROGRAM <identifier> ( <file identifier>
                                             { , <file identifier> } );

<file identifier> ::= <identifier>

<block> ::= <label declaration part> <constant definition part>
            <type definition part> <variable declaration part>
            <procedure and function declaration part> <statement part>

<label declaration part> ::= <empty> | LABEL <label> { , <label> } ;

<label> ::= <unsigned integer>

<constant definition part> ::= <empty> |
            CONST <constant definition> { ; <constant definition> };

<constant definition> ::= <identifier> = <constant>

<constant identifier> ::= <identifier>

<type definition part> ::= <empty> | TYPE <type definition>
                                       { ; <type definition> } ;

<type definition> ::= <identifier> = <type>

<type> ::= <simple type> | <structured type> | <pointer type>

<simple type> ::= <scalar type> | <subrange type> | <type identifier>
```

The BNF Description of Pascal (Cont'd.)

```
<scalar type> ::= ( <identifier> { , <identifier> } )

<subrange type> ::= <constant> .. <constant>

<type identifier> ::= <identifier>

<structured type> ::= <unpacked structured type> |
                                PACKED <unpacked structured type>

<unpacked structured type> ::= <array type> | <record type> |
                          <string type> | <set type> | <file type>

<array type> ::= ARRAY [ <index type> { , <index type> } ] OF
                                      <component type>

<index type> ::= <simple type>

<component type> ::= <type>

<record type> ::= RECORD <field list> END

<field list> ::= <fixed part> | <fixed part> ; <variant part> |
                 <variant part>

<fixed part> ::= <record section> { ; <record section> }

<record section> ::= <field identifier> { , <field identifier> } :
                                                   <type> | <empty>

<variant part> ::= CASE <tag field> <type identifier> OF
                                        <variant> { , <variant> }

<tag field> ::= <field identifier> : | <empty>

<variant> ::= <case label list> : ( <field list> ) | <empty>

<case label list> ::= <case label> { , <case label> }

<case label> ::= <constant>

<string type> ::= STRING | STRING[ <unsigned integer> ]

<set type> ::= SET OF <base type>

<base type> ::= <simple type>

<file type> ::= FILE OF <type> | TEXT | INTERACTIVE | FILE

<pointer type> ::= ^ <type identifier>

<variable declaration part> ::= <empty> | VAR <variable declaration>
                                   { , <variable declaration> } ;

<variable declaration> ::= <identifier> { , <identifier> } : <type>

<procedure and function declaration part> ::=
                             { <procedure or function declaration> ;}

<procedure or function declaration> ::= <procedure declaration> |
                                          <function declaration>

<procedure declaration> ::= <procedure heading> <block>

<procedure heading> ::= PROCEDURE <identifier> ; |
                  PROCEDURE <identifier>( <formal parameter section>
                               { ; <formal parameter section> } ) ;

<formal parameter section> ::= <parameter group> |
                                        VAR <parameter group>

<parameter group> ::= <identifier> { , <identifier> } :
                                             <type identifier>

<function declaration> ::= <function heading> <block>

<function heading> ::= FUNCTION <identifier> : <result type> ; |
                 FUNCTION <identifier> ( <formal parameter section>
                 { ; <formal parameter section> } ) : <result type> ;
```

The BNF Description of Pascal (Cont'd.)

```
<result type> ::= <type identifier>

<statement part> ::= <compound statement>


<statement> ::= <unlabelled statement> | <label> :
                                            <unlabelled statement>

<unlabelled statement> ::= <simple statement> | <structured statement>

<simple statement> ::= <assignment statement> | <procedure statement> |
                       <go to statement> | <empty statement>

<assignment statement> ::= <variable> := <expression> |
                           <function identifier> := <expression>

<variable> ::= <entire variable> | <component variable> |
               <referenced variable>

<entire variable> ::= <variable identifier>

<variable identifier> ::= <identifier>

<component variable> ::= <indexed variable> | <field designator> |
                                              <file buffer>

<indexed variable> ::= <array variable> [ <expression>
                                        { , <expression> } ]

<array variable> ::= <variable>

<field designator> ::= <record variable> . <field identifier>

<record variable> ::= <variable>

<field identifier> ::= <identifier>

<file buffer> ::= <file variable> ^

<file variable> ::= <variable>

<referenced variable> ::= <pointer variable> ^

<pointer variable> ::= <variable>

<expression> ::=<simple expression> |
     <simple expression> <relational operator> <simple expression>

<relational operator> ::=   = | <> | < | <= | >= | > | IN

<simple expression> ::= <term> | <sign> <term> |
                      <simple expression> <adding operator> <term>

<adding operator> ::= + | - | OR

<term> ::= <factor> | <term> <multiplying operator> <factor>

<multiplying operator> ::=   * | / | DIV | MOD | AND

<factor> ::= <variable> | <unsigned constant> | ( <expression> ) |
             <function designator> | <set> | NOT <factor>

<function designator> ::= <function identifier> |
                    <function identifier> ( <actual parameter>
                                          { , <actual parameter> } )

<function identifier> ::= <identifier>

<set> ::= [ <element list> ]

<element list> ::= <element> { , <element> } | <empty>

<element> ::= <expression> | <expression> .. <expression>

<procedure statement> ::= <procedure identifier> |
                    <procedure identifier> ( <actual parameter>
                                           { , <actual parameter> } )

<procedure identifier> ::= <identifier>
```

The BNF Description of Pascal (Cont'd.)

```
<actual parameter> ::= <expression> | <variable>

<go to statement> ::= GOTO <label>

<empty statement> ::= <empty>

<empty> ::=

<structured statement> ::= <compound statement> |
                           <conditional statement> |
                           <repetitive statement> | <with statement>

<compound statement> ::= BEGIN <statement> { ; <statement> } END

<conditional statement> ::= <if statement> | <case statement>

<if statement> ::= IF <expression> THEN <statement> |
                   IF <expression> THEN <statement> ELSE <statement>

<case statement> ::= CASE <expression> OF <case list element>
                     { ; <case list element> } END

<case list element> ::= <case label list> : <statement> | <empty>

<case label list> ::= <case label> { , <case label> }

<repetitive statement> ::= <while statement> | <repeat statement> |
                           <for statement>

<while statement> ::= WHILE <expression> DO <statement>

<repeat statement> ::= REPEAT <statement> { ; <statement> }
                       UNTIL <expression>

<for statement> ::= FOR <control variable> := <for list> DO
                                                      <statement>

<for list> ::= <initial value> TO <final value> |
                              <initial value> DOWNTO <final value>

<control variable> ::= <identifier>

<initial value> ::= <expression>

<final value> ::= <expression>

<with statement> ::= WITH <record variable list> DO <statement>

<record variable list> ::= <record variable> { , <record variable> }

<identifier> ::= <letter> { <letter or digit> }

<letter or digit> ::= <letter> | <digit>

<constant> ::= <unsigned number> | <sign> <unsigned number> |
               <constant identifier> | <sign> <constant identifier> |
               <string>

<unsigned number> ::= <unsigned integer> | <unsigned real>

<unsigned integer> ::= <digit> { <digit> }

<unsigned real> ::= <unsigned integer> . <digit> { <digit> } |
          <unsigned integer> . <digit> { <digit> } E <scale factor> |
          <unsigned integer> E <scale factor>

<scale factor> ::= <unsigned integer> | <sign> <unsigned integer>

<sign> ::= + | -

<string> ::= ' <character> { <character> } '

<unsigned constant> ::= <unsigned number> | <string> |
                                   <constant identifier> | NIL

<letter> ::=  A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|
              a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

<digit>  ::=  0|1|2|3|4|5|6|7|8|9|0
```

APPENDIX H SUMMARY OF EXTENSIONS TO STANDARD PASCAL

PASCAL-80 is an extension of Standard Pascal, as defined in the PASCAL User Manual, and incorporates structures and operations not found in Standard Pascal.

The following data types are extensions to Standard Pascal and are described in Chapter 2 and Chapter 7.

The string type
Untyped FILEs
INTERACTIVE files
The MESSAGE type

The following predeclared procedures and functions are extensions to Standard Pascal, and are described in Chapter 3:

```
blockread(f,a,b [,s]): integer
blockwrite(f,a,b [,s]): integer
bufferread(f,a,l [,b]): integer
bufferwrite(f,a,l [,b]): integer
close(f,PURGE)
concat(s,...,s): string
copy(s,i,l): string
delete(s,i,l)
errorset(p)
exit(p)
fillchar(a,l,c)
gotoxy(x,y)
insert(s,t,i)
ioresult: integer
length(s): integer
log(x): real
mark(p)
memavail: integer
moveleft(s,d,l)
moveright(s,d,l)
portinput(p,v)
portoutput(p,v)
pos(p,s): integer
pwroften(x): real
release(p)
reset(f,<string>)
rewrite(f,<string>)
scan(l,e,a): integer
seek(f,<integer>)
setpointer(p,v)
sizeof(v): integer
systemcall(o,a)
```

The following predeclared procedures and functions are extensions to PASCAL-80 specifically for RMX/80 applications:

```
rqacpt(mp,e)
rqctck
rqctsk(s)
rqcxch(e)
```

```
rqdlvl(1)
rqdtsk(t)
rqdxch(e): integer
rqelvl(1)
rqendi
rqgetlt(m,1,t)
rqisnd(i)
rqresm(t)
rqsend(e,m)
rqsetlt(m,1,t)
rqsetp(a,1)
rqsetv(a,1)
rqsusp(t)
rqwait(mp,e,t)
```

APPENDIX I IMPLEMENTATION DETAILS

Data Type Formats

Every variable in PASCAL-80 is aligned on an even-byte, or word, boundary, although individual elements of a PACKED structure may occupy as little as one bit (in a PACKED ARRAY OF boolean). All interpreter operations expect their operands to occupy at least one word, even if not all of the information in that word is valid.

boolean

One word, with bit 0 indicating the value (false=0, true=1). The other bits are ignored by the boolean operations.

char

One word, with the low byte containing the character. The high order byte is ignored by operations using a value of type char.

integer

One word. The value is stored in two's complement form, so that a variable of type integer can assume values in the range −32768..32767.

scalar

One word, with value in the range 0..32767.

real

Two words, in INTEL floating point format.

POINTER

One word, with a value in the range 0..65535.

SET

0..255 words, depending upon the number of elements in the set. Sets are stored as bit vectors, with a lower index of zero, where each bit corresponds to a possible set element. A set variable declared as "SET OF a..b" is allocated (b+15)div16 words, all of which contain valid information. All elements past the last word of a set are assumed to not be elements of the set.

RECORDs and ARRAYs

Any number of words, depending upon the size of the structure. Arrays are stored in row-major order. When a record or an array is an operand to a procedure or internal operation, the structure itself is never loaded onto the stack; just a reference (pointer) to it.

string

1..128 words, depending upon the declared maximum length of the string. A variable declared as "s: string[n]" is allocated (n div 2)+1 words. The first byte of the string (s[0]) contains the current valid length of the string, while the bytes 1..s[0] contain the valid characters.

Implementation Size Limits

- The maximum number of characters in a variable of type string is 255.
- The maximum number of elements in a set is 255*16=4080.
- The maximum number of PROCEDUREs or FUNCTIONs in a segment is 127.
- The maximum number of user SEGMENT PROCEDUREs and SEGMENT FUNCTIONs is 7.
- The maximum number of bytes of object code in a PROCEDURE or FUNCTION is 1200.
- The maximum size of a data frame is 16383 words.

Differences Between PASCAL-80 and Standard Pascal

In PASCAL-80, the **goto** statement may only reference local labels; i.e., labels declared in the same block as the **goto** statement itself. A **goto** statement may not reference a label declared in an enclosing block, as is possible in Standard Pascal. However, the predeclared procedure **exit** overcomes this limitation and provides even more capabilities.

PASCAL-80 does not support the standard procedures **pack** and **unpack** as described in the PASCAL User Manual.

PASCAL-80 limits the (enumerated) elements of **a** set to positive values only.

PASCAL-80 does not support the construct in which PROCEDUREs and FUNCTIONs may be declared as parameters of a PROCEDURE or FUNCTION.

PASCAL-80 does not support the predeclared procedure **dispose**. However, the procedures **mark** and **release** can be used to approximate the actions of **dispose**.

APPENDIX J
PASCAL REFERENCE TEXTS

The primary reference text for Pascal-80, the source which defines Standard Pascal, is:

PASCAL User Manual and Report
Kathleen Jensen and Niklaus Wirth
Springer-Verlag, New York, © 1974
Corrected Printing, 1978

In addition, the following texts should be investigated by anyone interested in programming in Pascal:

Algorithms + Data Structures = Programs
Niklaus Wirth
Prentice-Hall, © 1976

A Practical Introduction to Pascal
I.R. Wilson & A.M. Addyman
Springler-Verlag, New York, © 1979

PASCAL, An Introduction to Methodical Programming
W. Findlay & D.A. Watt
Computer Science Press, © 1978

(Microcomputer) Problem Solving Using PASCAL
Kenneth L. Bowles
Springer-Verlag, New York © 1977

Programming in PASCAL
Peter Grogono
Addison-Wesley, © 1978

An Introduction to Programming and Problem Solving with PASCAL
Schneider, Weingart, Perlman
John Wiley & Sons, © 1978

Structured Programming and Problem-Solving with PASCAL
Richard B. Keiburtz
Prentice-Hall, © 1978

APPENDIX K DIRECTORY OF RELEASE DISKETTES

The PASCAL-80 System software is contained on each of two (2) release diskettes (single and double density). The directory listing of each of these diskettes is as follows:

Single Density Release Diskette

```
DIRECTORY OF :F4: 9500065.02
NAME   .EXT  BLKS   LENGTH ATTR        NAME   .EXT  BLKS   LENGTH ATTR
PASCAL         81    10053             PASCAL.RES   102    12800
COMP   .COD   293    36864             JOIN   .COD   25     3072
GENOBJ.COD     17     2048             P80EXT.LIB     2      127
P80RUN.LIB    378    47569             P80ISS.LIB   236    29588
P80RAR.LIB    205    25603             P80ISS.PLB   238    29843
EX     .PAS     8      810             BUFFER.PAS     8      878
ERROR  .PAS     8      847             SEEKEX.PAS    49     6120
PEOPLE.DAT      7      700             FIG31  .PAS    8      869
FIG32  .PAS     9      982             FIG33  .PAS   10     1149
FIG34  .PAS    12     1349             FIG35  .PAS    4      366
FIG36  .PAS    13     1480             FIG37  .PAS    8      811
FIG38  .PAS    10     1152             SIX3   .PAS    6      544
ASUM   .OBJ     2      100             ASUM   .LST   15     1714
ASUM   .ASM     7      733             PSUM   .PLM    4      361
PSUM   .OBJ     3      130             PSUM   .LST   22     2585
SIX3   .CSD     2      111             SIX1   .PAM    6      576
SIX1   .PA2     4      347             SIX1   .PA1    4      279
SIX1   .CSD     3      193             EIGHT1.PAS     8      795
                                1817
1872/2002 BLOCKS USED
```

Double Density Release Diskette

```
DIRECTORY OF :F1:970058.02
NAME   .EXT  BLKS   LENGTH ATTR        NAME   .EXT  BLKS   LENGTH ATTR
PASCAL         81    10053             PASCAL.RES   102    12800
COMP   .COD   293    36864             JOIN   .COD   25     3072
GENOBJ.COD     17     2048             P80EXT.LIB     2      127
P80RUN.LIB    378    47569             P80ISS.LIB   236    29588
P80RAR.LIB    205    25603             P80ISS.PLB   238    29843
EX     .PAS     8      810             BUFFER.PAS     8      878
ERROR  .PAS     8      847             SEEKEX.PAS    49     6120
PEOPLE.DAT      7      700             FIG31  .PAS    8      869
FIG32  .PAS     9      982             FIG33  .PAS   10     1149
FIG34  .PAS    12     1349             FIG35  .PAS    4      366
FIG36  .PAS    13     1480             FIG37  .PAS    8      811
FIG38  .PAS    10     1152             SIX3   .PAS    6      544
ASUM   .OBJ     2      100             ASUM   .LST   15     1714
ASUM   .ASM     7      733             PSUM   .PLM    4      361
PSUM   .OBJ     3      130             PSUM   .LST   22     2585
SIX3   .CSD     2      111             SIX1   .PAM    6      576
SIX1   .PA2     4      347             SIX1   .PA1    4      279
SIX1   .CSD     3      193             EIGHT1.PAS     8      795
                                1817
1926/4004 BLOCKS USED
```