10-0000009-5

# MONROE PASCAL
# PROGRAMMER'S REFERENCE MANUAL

June 1982

Rev. 1

MONROE SYSTEMS FOR BUSINESS
The American Rd.
Morris Plains, N.J. 07950



2139-S (Rev. 1)

## PURPOSE OF THIS DOCUMENT

This document is a Programmer's Reference Manual. It is to be used by experienced programmers as a reference tool. It is not intended for use as a learning aid by non-programmers.

## RECORD OF CHANGES

| Change No. | Date | Pages Affected | Description of Changes |
|---|---|---|---|
| | 11/81 | | Preliminary Edition |
| A | 5/82 | Sections 14-17<br>Appendices G-L | New Information Added |
| Rev. 1 | 6/82 | | Manual reprinted (includes Change A) |

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Cont.)

LIST OF TABLES

# SECTION 1
# INTRODUCTION

1

# SECTION 1
# INTRODUCTION

## 1.1 INTRODUCTION TO PASCAL

PASCAL is a relatively new language that has been accepted and implemented worldwide. It was first published in 1971, yet already it is considered one of the most promising problem-solving languages available.

PASCAL has two powerful strengths that account for its popularity. The first is that it is one of the few languages that was designed for structured programming, a method of writing programs that is sequential and well-ordered. It permits the programming of extremely large and complex projects while minimizing the debugging time.

The second strength is that PASCAL has a small but very powerful set of commands that makes it a relatively easy language to learn and to use. It was also designed to be completely machine-independent so PASCAL programs are transportable and maintainable. It is even more flexible because it facilitates the defining of complex data structures specifically for each application.

PASCAL is a compiled language. This means that a PASCAL program is completely translated into object code before it can be executed. Therefore, it is not interactive in the sense that BASIC and other interpreted languages are.

Monroe's PASCAL language described in this manual is standard PASCAL. It is available on Monroe's educational and occupational 8800 computer series.

## 1.2 ABOUT THE MANUAL

The PASCAL PROGRAMMER'S REFERENCE MANUAL is designed to be just that: a reference manual for an experienced programmer, not a tutorial. Although it is not designed for those learning PASCAL, many examples are included to help you understand and implement the language.

## Text Symbols and Conventions

This manual uses specific documentation conventions to describe all PASCAL statement, function, procedure, and command formats. These conventions are as follows:

**Symbol** — **Description and Use**

1. CAPITAL LETTERS

Capital letters are used for all keywords, standard functions and procedures, and commands that are to be explicitly typed.

Examples: BEGIN
WRITELN

2. Lower case

Lower case letters specify variables to be supplied by the user according to the rules explained below and in this text.

Examples: <identifier>: = <constant>;
WHILE <boolean expression>

3. <>

Angled brackets enclose fields that are required for valid syntax. The brackets are never to be typed.

Examples: IF <boolean expression>
THEN <statement>;

4. | |

Vertical lines enclose optional elements of a statement. When a statement contains more than one optional element, each may be underlined to clarify any resulting ambiguities. (See item 6.)

Examples: REPEAT
IF <condition>
THEN <statement block>
|ELSE statement block|

| Symbol | Description and Use |
|---|---|
| 5. [](),.; | Square brackets and parentheses enclose required elements or keywords of a statement. Commas are separators. Periods and semicolons are delimiters. They must all be typed exactly as shown.<br><br>Examples: ARRAY[<const>..<const>]<br>would be: ARRAY[1..10]; |
| 6. ... | Ellipses (3 dots) indicate that multiple arguments are allowed.<br><br>Example: READ (\|fd\|,<ident>\|,ident,...\|) |

Organization of the Manual

This manual is organized into 17 sections and twelve appendices.

Section 1 gives a general overview of this document.

Section 2 gives machine-specific information about running PASCAL. All programmers should read this section carefully.

Sections 3 through 5 contain information regarding some of the more basic identifiers in the PASCAL language.

Sections 6 through 11 describe individual commands, more advanced identifier definitions, and program and function definitions. The statements will each be explained and summarized in the following format:

1. Function -Summarizes the purpose of the statement.
2. Format -Shows the statement syntax.
3. Arguments -Defines the format variables.
4. Use -Describes where and under what circumstances the statement would be used.
5. Note -Important exceptions and limitations.
6. Example -Illustrates various uses of the command.

Section 12 describes the PASCAL intrinsics.

Section 13 details PASCAL's system commands and the options that are available to the user.

Section 14 describes the statements used to load and modify ISAM data files.

Section 15 describes low resolution business graphics.

Sections 16 and 17 deal with low and high resolution color graphics, respectively.

Appendix A summarizes the standard functions and procedures that are available.

Appendix B explains the possible compile-time options.

Appendices C and D list the compile-time and run-time errors, respectively.

Appendix E summarizes the operators, their uses, and their operands, while Appendix F lists all the legal characters.

Appendix G contains sample programs.

Appendix H shows the low resolution color graphics character set.

Appendix I contains the high resolution color selection chart.

Appendix J shows the resulting background color when high and low resolution color graphics are displayed on the screen simultaneously.

Appendix K lists the port numbers and associated devices.

Appendix L describes the RLDR Utility which is used to build an executable program (i.e., Task).

## Abbreviations

The following abbreviations are used in this manual:

| | |
|---|---|
| cfd | File descriptor of the command file |
| const | Constant |
| fd | File descriptor |
| ident | Identifier |
| infd | File descriptor that contains the PASCAL p-code |
| lfd | File descriptor for the list file |
| libfd | File descriptor for the p-code library |
| outfd | File descriptor that contains the relocatable object file |
| stmt | Statement |
| tid | Name assigned to the task (four letters) when it is loaded into memory |
| var | Variable |

## 1.3 FILE-VOLUME-DEVICE-NAMING CONVENTIONS

The Monroe Operating System file, volume, and device naming conventions are defined as follows:

A) A file is a program or a collection of data stored on a disk-type storage medium. Once saved files stay on the disk permanently unless they are explicitly removed.

B) A volume name is a name given by the user to a disk. Filenames must be preceded by their volume name unless they reside on the system volume. The system volume is the volume from which the operating system is booted. It can be reset by the user.

C) A device name is a name given to a physical device (e.g., CON: for the console, PR: for the printer, FPY0: for drive 0 (lower drive), FPY1: for drive 1 (upper drive). These names cannot be changed by the user.

D) File descriptors, hereafter referred to as "fd" in this manual, can be composed of four fields: vol, filename, directory, and type, where vol can be either a volume or when used alone as a device name. Device descriptors are composed of the device mnemonic only.

E) The format can be expressed in four ways:

1. <device:>
2. [vol:]<filename>[/type]
3. [vol:]<directory>
4. [vol:]<directory.filename>[/type]

where:

| | |
|---|---|
| vol/ device | Vol is the name of the disk on which the file resides it the file descriptor refers to a file, or the name of a device if the file descriptor refers to a device. It may be from one to four characters. The first character must be alphabetic and the remaining alphanumeric. If the volume is not specified, the default volume is the SYSTEM volume. |
| filename | Name of the file. It may be from one to twelve alphanumeric characters. |
| directory | Name of the element file directory. It may be from one to twelve alphanumeric characters. If not specified, the directory defaults to the Master File directory. |
| type | Type of file, i.e., A=ASCII, B=Binary, etc. |

Example: Examples of legal file/device descriptors are:

| | |
|---|---|
| -EDIT MONT:REPORT | Edits file REPORT on the volume MONT. |
| PROGRAM HELP(tst50,tst60) | Files tst50 and tst60 will be used in the PASCAL program named HELP. |
| -PASSYS PASCOMP,VOLA:HELP | Compiles file HELP on volume VOLA. |
| -PASCAL VOLA:HELP | Executes the compiled program HELP on volume VOLA. |
| PROCEDURE WRITELN (TESTPRG:text file) | Writes the value to the file TESTPRG and then inserts a carriage return character. |

## 1.4 RELATED MANUALS

This manual is as self-sufficient as possible. However, instructional information about the Utilities and the Text Editor may be required to effectively use the PASCAL package. For additional information, refer to the following 8800 Series Programmer's Reference Manuals:

- Utility Programs
- Text Editor
- Monroe Operating System

# SECTION 2
# WORKING WITH PASCAL

1

SECTION 2

WORKING WITH PASCAL

## 2.1 OVERVIEW

Monroe Pascal software for the 8800 Series Computers is delivered on a disk containing a compiler PASCOMP, two interpreters PASSYS and PASCAL, and supplementary system programs. (Each is described in detail in Section 13.)

The following Pascal system programs are written in native code for the Monroe computer: PASSYS, PASCAL, PASOBJ, RLDR, and PASRTL. All others consist of "psuedo-code" which can be interpreted by the PASSYS interpreter; PASSYS thus constitutes the basis for most Pascal-related operations. User written application programs can be translated by the PASCAL system either to pseudo-code or to the native code of the computer. PASCAL is used to interpret user-written programs which have been translated to pseudo-code; alternatively, PASRTL and RLDR (see Appendix L) can be used to convert programs which have been translated to native code into directly executable task files.

## 2.2 DISK HANDLING

In order to use your Pascal disk, certain procedures must be followed. Shown below is one method that can be used. However, there are other methods which may be more efficient, depending on your knowledge of the system.

Dual Drive OC System Procedure:

1. Boot from your MS8 disk in drive 0 (lower drive).

2. Put PASCAL disk in drive 1 (upper drive), and open the drive--OPEN FPY1:.

3. Copy necessary utilities and system programs from your MS8 disk to the PASCAL disk (PASC:) which may be needed later. Copy, for example: EDIT, CMD$VOLUME, CMD$LIB, ISAM, COPYLIB, etc. Once this has been done, there is no need to do this in subsequent sessions.

4. Change system volume to PASC: (PASCAL disk) using the volume utility--V PASC:.
5. Remove MS8 disk and insert and open a data disk previously initialized.

The text editor - EDIT can now be used to create a PASCAL source program. (See Section 2.5.)

NOTE: If you want your source program to be stored on the data disk, prefix the filename with the volume name, e.g., DATA:SOURCEFILE.

Single Drive EC System Procedure:

1. Boot from your MS8 disk.
2. Execute COPYF Utility (refer to the 8800 Series Monroe Utility Programmer's Reference Manual) to copy necessary utilities and system programs from your MS8 disk to the PASCAL disk-PASC:. Copy, for example: EDIT, CMD$VOLUME, CMD$LIB, ISAM, COPYLIB, etc. Once this has been done, there is no need to do this in subsequent sessions.
3. Remove System disk (assume step 2 has been done). Insert and open PASCAL disk--OPEN PASC:.
4. Change system volume to PASCAL disk--PASC:V PASC:.

You are now ready to use the Editor to create a source program.

## 2.3 WORD LENGTHS FOR FILE AND PROGRAM IDENTIFIERS

Identifiers are alphanumeric words that have specific meanings much like words in informal languages. They are used to define constants, types, variables, procedures and functions, and files. PASCAL allows identifiers of any length provided that they do not span more than one line. This allows meaningful names to be used for all identifiers, hence, the program can be read much easier. It is important to note that only the first eight characters are significant, i.e., "newgraphx" and "newgraphy" are both valid identifiers but are indistinguishable to the compiler; "xnewgraph" and "ynewgraph" might be used instead.

The first character in an identifier must be a letter; the remaining characters may be either letters or digits. All other ASCII characters are illegal. Also, no reserved words (see Section 3.1 for a list) may be used.

<u>Examples:</u>

The following identifiers are all legal:

abc, time2, C8915, Idname

The following identifiers are illegal for the reasons stated:

| | |
|---|---|
| 4aname | does not start with a letter. |
| b+c | + is not a legal character. |
| the name | a blank is not a legal character. |
| var | VAR is a reserved word. |

## 2.4 PASCAL PROGRAM SYNTAX

PASCAL programs consist of a heading and a block section. The general format is:

```
PROGRAM<name>|(fd,...)|;
  |Declarations|
  <compound statement>.
```

<name> is the identifier for the program. The optional list of filenames designates the files to be used in the program. They must be declared in the Variable Declarations section.

The Declarations section is composed of the following parts:

```
|label declarations|
|constant definitions|
|type definitions|
|variable declarations|
|procedure and function declarations|
```

These parts must exist in the order that they are listed above. They will be described in greater detail in Sections 2 through 11.

## 2.5 WRITING A PROGRAM

Since PASCAL programs must exist in text files, all programs are created and manipulated through the Text Editor--Edit. Enter the following command to invoke the Editor:

```
EDIT <fd>
```

The fd is the file descriptor as defined in Section 1.4. Each time this command is entered with a new filename, a file is created and the name is placed in the disk's file directory. If an existing filename is used, the Editor is invoked and the existing file is opened.

Next, the contents of the file must be read into the buffer where it can be manipulated. This is done by executing the Read (RE) command. This command should be used exactly once each time the editor is invoked or else the file and the buffer will be lost. If the file is just being created, the RE command will return a length of zero. Otherwise, it will give the length of the program existing in the file.

It is often a good idea to look at the file even if it was just created to be sure that it has not accidentally been used before. The Print (PR) command accomplishes this. If the listing is longer than the CRT display, the first section is displayed and the lines that follow may be seen by hitting the space bar. To exit the Print command, press the RETURN key.

The Insert Line (IL) is used to begin entering the program. A line number will appear at the left of the screen with the cursor following it. To exit the Insert Line command, type a "#" in the first position on the line.

The Output and READ (OR) command can be used to load the next section of a large program file until the end of the file is reached.

The following Text Editor commands are available:

| Command | Function |
|---|---|
| AB | Abort session. |
| BT | Set tab stops. |
| CV | Change variable. |
| DL | Delete line(s). |
| ED | Edit line. |
| EN | Normal termination. |
| IL | Insert line(s). |
| KI | Kills the buffer, the file, and the backup file. |
| LC | Enable lower case input. |
| NU | Renumber. |
| OR | Output, kill the buffer, and read. |
| PR | Print. |
| RE | Kill the buffer and read. |
| SV | Search for string variable. |
| UC | Force input to upper case. |
| WR | Write current buffer. |

For a more complete description of the uses and parameters of these commands, refer to the 8800 Series Text Editor Programmer's Reference Manual.

When an editing session is completed the End (EN) command will terminate the session, write the buffer to the disk file, and exit the Editor.

## 2.6 COMPILING A PROGRAM

Once the program has been written and the editing session is ended, the program must be interpreted into a pseudo code program so that it can be executed. To do this, the compiler routine must be called. The simplest forms of this command are:

| | |
|---|---|
| PASSYS PASCOMP,<fd> | Compiles program and displays information on the console indicating when a block is being compiled and error messages if any. |
| PASSYS,,10000ƀ,<fd>,,CON:<br>(ƀ = required blank) | Displays complete program with line numbers as is being compiled to the console including error messages if any. |

There are a series of options that may be set if necessary. Refer to Section 14 for the more complex versions.

The compiler stores the p-code program it produces in a file it creates using the same filename but with a file type of BP-BINPAS. The source program is A-ASCII.

The compiler will flag all syntactic and semantic errors, known as compile time errors. After compilation, the programmer must return to the Editor to correct these errors. Refer to Appendix C for the list of compile time errors and their codes.

Note: It is important to remember that the program must be recompiled after each editing session for the changes to be reflected in the object file.

## 2.7 RUNNING A PROGRAM

Once the program has been successfully compiled, the program is ready for execution. The format for the simplest version of this command is:

PASCAL <fd>

There are switches and other options available for more advanced users which are discussed in Section 13.

An error will be displayed if no p-code file with the given filename is found.

Run time errors will be displayed if there are inconsistencies in the logic of the program. See Appendix D for the list of these error codes and their meanings. Once again, the Editor must be invoked to correct run time errors in the program. To see run time errors with the number, a compile option must be set. (See D switch option, Appendix B.)

The program may be manually interrupted using the CONTROL-A which is executed by holding down the Control key and typing an "A". It will be cancelled if the CONTROL-A command is followed by the Cancel (CA) command. The "End of Task" appears to signal that the termination route has been completed.

## 2.8 BAUD RATE SELECTION

The system default printer baud rate is 1200 Bd. The baud rate is selectable at run time by defining a file-descriptor beginning with "PR:Rx" where "x" defines the baud rate, as follows:

0 = 75 Bd
1 = 110 Bd
2 = 300 Bd
3 = 600 Bd
4 = 1200 Bd
5 = 2400 Bd
6 = 4800 Bd
7 = 9600 Bd
8 = 19200 Bd

Example: To specify a baud rate of 2400 for example, a PASCAL source program must include the following statements:

```
"<variable name>:='PR:R5';"
"RESET(<text filename>,<variable name>);"
```

This program must be executed to set the baud rate.

Note: The baud rates specified must be compatible with the speed of the printer; otherwise, erroneous results will occur.

# SECTION 3
# SPECIAL SYMBOLS AND CONSTANTS

## SECTION 3
## SPECIAL SYMBOLS AND CONSTANTS

### 3.1 IDENTIFIERS

The PASCAL vocabulary is made up of basic symbols categorized as letters, digits, and special symbols. Special symbols are operators, delimiters, and reserved words. Delimiters and reserved words are interpreted as single symbols with specific meanings.

Although identifiers may be 32 characters long, only the first eight are significant.

Identifiers are combinations of letters and digits that define constants, types, variables, and procedures and programs. They were introduced in Section 2.1.

#### Reserved Words/Special Symbols

There are some identifiers and symbols that have specific meanings in PASCAL and cannot be used in any other way. The following is a list of reserved words:

| | | | |
|---|---|---|---|
| AND | EXTERNAL | NEW | REPEAT |
| ARRAY | FILE | NIL | RETURN |
| BEGIN | FOR | NOT | SEGMENT |
| BOOLEAN | FORWARD | OF | SET |
| CASE | FREE | OR | STRING |
| CHAR | FUNCTION | OVERLAY | TEXT |
| CHR | GOTO | PACKED | THEN |
| CONST | GOTOXY | PROCEDURE | TO |
| DIV | IF | PROGRAM | TYPE |
| DO | IN | PUT | UNTIL |
| DOWNTO | INCLUDE | READ | VAR |
| ELSE | INTEGER | READKEY | WHILE |
| END | ISAMFILE | READLN | WITH |
| ENTRY | LABEL | REAL | WRITE |
| EXIT | MOD | RECORD | WRITELN |

In addition all standard function and procedure names are reserved (see Sections 11 and 12). Incorrect usage of reserved words will cause errors. Hence, the meaning and function of each should be checked before use.

The following symbols and groups of symbols have special meanings and cannot be used as part of user-defined identifiers.

| | | | |
|---|---|---|---|
| + | ; | > | { |
| - | " | <= | } |
| * | : | >= | ^ |
| / | , | ( | .. |
| := | = | ) | (* |
| . | <> | [ | *) |
| , | < | ] | |

The meanings and uses of these are detailed in Appendix E.

User-Defined Words

All constants, types, variables, procedures, programs, and files must be defined and an identifier associated with it so that it may be used. These are called user-defined identifiers. When a command format in this manual contains an identifier field, it refers to a user-defined identifier.

These identifiers must follow certain rules:

1. They may be any length but must be able to fit on one line.

2. Only the first eight characters are significant when differentiating between identifiers.

3. They must begin with a letter of the alphabet.

4. The remaining characters may only be letters or digits. All other symbols are illegal.

5. No reserved words may be used.

Refer to Section 2.1 for examples of valid and invalid identifiers.

## 3.2 NUMBERS

PASCAL has the facility to represent base ten numbers as either integers or reals. They may be positive, negative, or zero.

### Integers

Integers may be thought of in the everyday sense. 93 and -245 are integers while 1.92 and -3.1417 are not. The integers may range from -32768 to +32767. The positive number 32767 is kept as a system identifier under the name maxint. Maxint may be referred to directly in a program. For example:

```
CONST      max = maxint;

FOR  counter := 1 TO maxint DO WRITELN(counter);
```

### Reals

Real numbers have an integer part and a decimal part. They can be represented the way they usually are in mathematics (i.e., 345) or using exponential notation. Exponential notation has a decimal number portion and a scale factor. The letter E precedes the scale factor and means "times ten to the power of". If the decimal portion contains a decimal point, at least one digit must precede and one succeed the point. The scale must be between 2.93874E-39 and 1.70141E+37 or the value defaults to zero. There is seven-digit accuracy for default.

Some valid examples are:

| | |
|---|---|
| 11E4 | 110000.0 |
| -1.35 | -135.0 |
| 21.55E-3 | 0.02155 |
| 1.93 | 1.93 |

Some invalid ones are:

| | |
|---|---|
| .92E1 | no digit preceding the decimal point. |
| 1.E1 | no digit following the decimal point. |
| E5 | no mantissa. |
| 2.3E | no exponent. |
| 5.1E1.5 | fraction in the exponent. |

## 3.3 STRING CONSTANTS

Strings are sequences of characters enclosed in single quotes. They are often used for text and for titles, headings and comments in output. Any character may appear within the string. However, if a single quote is needed, two single quotes in a row must be used.

Examples:

'Age'   'Title'   '405 Makalapa Drive'   'C'

## 3.4 COMMENTS

It is always important to document any program so that it can be read and easily understood. This is especially important if it may be used by others, or even if it may be used by the programmer a long time after it was written.

Comments may be included anywhere in a program, though they usually appear to the right of the line of code they discuss. They may be removed from the program anytime without affecting the program.

A comment has the following structure:

(* <any sequence of characters/symbols except '*)'> *)

The (* and *) may be replaced by { and } respectively.

Comments may be inserted or deleted from a source program without affecting the P-code, unless the comment contains executable source codes requiring the user to recompile the source program.

| | |
|---|---|
| (*X:=7;*) | This line is not translated to P-code: |
| X:=7; | Removing the (*and*) will require a new compilation. This method is useful for debugging a PASCAL program. |

# SECTION 4
# PROGRAM HEADINGS AND DECLARATIONS

SECTION 4

PROGRAM HEADINGS AND DECLARATIONS

## 4.1 PROGRAM HEADING

All programs in PASCAL must have a heading and a block. The heading gives the program its name and lists all the files it uses. The general format for a program is:

```
PROGRAM <name>|(fd,...)|;  (*";" is a statement separator.*)
|Declarations|
<Compound Statements>.     (*"." marks the end of the program.*)
```

<name> is the program name. The optional list of filenames designates the files to be used.

The Declaration section is composed of the following sequence:

```
|label declarations|
|constant definitions|
|type definitions|
|variable declarations|
|procedure and function declarations|
```

Note: Sequence errors will result if this order is not followed.

The Compound Statement is:

```
BEGIN
  <statement>;
  |statements;...|
END.
```

Example: PROGRAM getchr (readline,printline);

## 4.2 LABEL DECLARATIONS

Referring back to Section 2.1, a program consists of a heading and a block. The block contains a declaration part where all identifiers local to the program are defined. The first section of this part is where the labels are declared.

A label is a prefix to a statement so that it can be referenced elsewhere in the program by a GOTO statement. Any statement except the conditional parts of IF and CASE statements may be labeled. The label must be defined as an unsigned integer consisting of at most four (4) digits. The format is:

```
LABEL          <label>|,label,...|;
```

Example: `LABEL 4, 931, 4444;`

A statement is labeled according to the following format:

```
<label>: <statement>;
```

Example: `5: READ (testvalue);`

If no labels are needed the LABEL declaration part is completely omitted.

## 4.3 CONSTANT DEFINITIONS

A program sometimes uses a value that remains unchanged throughout its execution, such as Pi or MAXINT. These values are defined as constants and assigned to identifiers so they may be referenced throughout the program. This makes a program more readable and is considered a good documentation practice. The format for the definition is:

```
CONST          <ident>=<const>;
               |ident=const;...|
```

The identifiers may be any legal user-defined identifier. The constant values may be numbers, constant identifiers, or strings. As many constants as the program needs can be defined. If none are needed, the CONST definition part is completely omitted.

If an identifier that has been defined as a constant is assigned a new value in the program, then a compile time error will occur. Once the identifier has been defined, it can only refer to that value.

Some examples of constant definitions are:

```
Ex. 1         CONST      valint=MAXINT;
                         maxnumpeople = 500;
                         feed = '(:12:)'
Ex. 2         CONST      pi = 3.14;
                         cardlen = 80;
                         linelen = 132;
```

## 4.4 TYPE DEFINITIONS

There are some standard data types which have already been mentioned. These are INTEGER, REAL, CHAR, STRING and BOOLEAN. However, PASCAL has the capability of declaring more abstract types with the user defining the properties associated with them. These types may be scalar, subrange, set, array, record, file, and pointer enumerated types. Each of these types will be discussed in depth in Sections 7 through 10. However, the general form of the TYPE definition part is:

```
TYPE          <ident> = <type declaration>;
              |ident = type declaration;...|
```

The identifiers may be any legal PASCAL user-defined identifier. The format and legal elements of the type declaration field varies with different types so they will be discussed later where appropriate.

If no user-defined types are needed, the TYPE definition section is omitted. If included, it must be placed in the correct sequence (i.e., before VAR).

Examples:

These examples show enumerated type.

```
Ex. 1     TYPE     days = (Sunday, Monday, Tuesday, Wednesday,
                          Thursday, Friday, Saturday);

Ex. 2     TYPE     text = (True, False, Undecided);
                   digit = 0..9;
```

## 4.5 VARIABLE DECLARATIONS

Every variable that occurs in a program must first be defined in the variable declaration part. The format for the variable declarations is:

```
VAR          <ident>|,ident,...| : <type>;
             |ident,... : type;|
```

The identifiers may be any legal PASCAL identifier. The types may be INTEGER, REAL, CHAR, BOOLEAN, STRING or any type defined in the TYPE definition part. These variables may be assigned new values within the program. If no variables are needed, the section is completely omitted. If included, it must be placed in the appropriate sequence (i.e., before procedures or functions).

Examples:

Ex. 1

```
VAR     count, intval : INTEGER;
        sum, realval : REAL;
```

Ex. 2

```
VAR     answer : test;
        number : digit;
        counter,index : INTEGER;
```

## 4.6 PROCEDURE AND FUNCTION DEFINITION

Programs often require that sections of the code appear in more than one place in the program. If, for example, a twenty-five line section was needed in five different places, there would be one hundred lines of redundant code. Instead, the code could be put into procedures or functions that would then be called by the program. Procedures and functions are like subroutines in that they can be called by the main program and by each other. However, before they can be called, they must be declared and defined. This section comes after the variable declaration part of the block of the main program.

The composition of functions and procedures is the same as a program. They have headings and blocks that are of the form:

```
PROCEDURE <name>|(parameter list)|;
  |declarations|
  BEGIN
    <statement block>
  END;

FUNCTION <name>|(parameter list)|:<type>;
  |declarations|
  BEGIN
    <statement block>
  END;
```

This is covered in more detail in Section 11 but there is one important fact to remember: procedure or function must be declared before it is used. For example, if the main program calls FUNCTION A, which in turn calls PROCEDURE B and PROCEDURE C, the B and C must be defined before A is. The correct order is:

```
PROGRAM   main;
  VAR     val : INTEGER;

PROCEDURE B;
BEGIN
     :
END; (*B*)

PROCEDURE C;
BEGIN
     :
END; (*C*)
```

```
FUNCTION A;
BEGIN
    B;  (* call B *)
    C;  (* call C *)
END; (*A*)

BEGIN
    val:=A; (* call A *)
END. (* main *)
```

Forward references are covered in Section 11.

# SECTION 5
# CONTROL STATEMENTS

## SECTION 5
## CONTROL STATEMENTS

### 5.1 INTRODUCTION

Control statements describe the actions that a program is to perform on its defined data. Together these statements form the statement part of a program. Between every two statements there must be a semicolon that acts as a statement separator and that is not considered to be part of either statement.

Monroe PASCAL statements that are available to a user are summarized in Table 6-1. Each is explained in greater detail in this section.

Table 5-1. PASCAL Statements

| Statement | Description |
| --- | --- |
| <a>:=<e> | Assigns values to variables. |
| BEGIN | Sets off a compound statement. |
| END | Terminates a compound statement. |
| WHILE | Executes a statement or compound statement repeatedly using a leading decision. |
| REPEAT | Executes a loop repeatedly using a trailing decision. |
| FOR | Executes a loop a predetermined number of times. |
| IF | Evaluates an expression and performs one of two possible actions. |
| CASE | Transfers control to one of several statement labels depending on the variables value. |
| GOTO | Unconditionally transfers control from one portion of a program to another. |

## 5.2 COMPOUND STATEMENT

A compound statement is a sequence of statements that are set off by the reserved word BEGIN before the first statement and by END after the last. Simple statements may be extended with additional instructions using a compound statement structure. This structure allows nested compound statements. The format for a compound statement is:

```
If A=4 THEN
  BEGIN
    statement1;
    statement2
  END;
```

The WHILE and FOR statements discussed in this section contain examples of compound statements.

## 5.3 ASSIGNMENT STATEMENT

Function: To assign values to variables.

Format: <identifier>:=<expression>;

Arguments: The identifier may be any user-defined identifier. The expression may be a user-defined identifier that has an assigned value, a constant, or a mathematical expression using arithmetical, relational or logical operators. The identifier type must match the expression type.

Note: The identifier takes on the value of the expression.

Use: Although it is used simply to assign a value to a variable, it is used more often as a way to evaluate an expression and retain the result as the value associated with the user-defined identifier.

Note: The convention of evaluating expressions from left to right using operator precedence is observed within the expression. The operators below are ranked according to precedence with NOT having the highest and the relational operators having the lowest. Those on the same line have equal precedence values.

```
( )
NOT
*, /, DIV, MOD, AND
+, -, OR
=, <>, <, <=, >=, >, IN
```

If an expression is enclosed in parentheses it is evaluated independently of the preceding and succeeding operators.

**Examples:**

| Expression | | Equivalent | | Result |
|---|---|---|---|---|
| 16 DIV 3 * 9 | = | (16 DIV 3) * 9 | = | 45 |
| 4 * 9 - 8 * 4 | = | (4 * 9) - (8 * 4) | = | 4 |

All data types in an expression must be compatible.

Declaration:

```
VAR         value, count, nextletter, length,
            side1, side2 : INTEGER;
            character : CHAR;
```

Main Section:

```
BEGIN
  value := 1;
  count := count+1;
  character := chr(nextletter+1);
  length := 2*(side1 + side2);
END. (* END PROGRAM *)
```

## 5.4 REPETITIVE STATEMENTS

Some programs require a set of statements be executed more than once. These statements form what is called a loop or iteration. Since the loop must be executed a finite number of times, a decision whether or not to continue executing the statements inside the loop must be made during each execution of the loop. This decision can be made at the beginning of the loop, called a leading decision (WHILE statement), or at the end of the loop, called a trailing decision (UNTIL statement). The FOR statement is used when the number of repetitions is a numeric value that can be computed. It also allows the program to keep an index variable available to the user.

The following repetitive statements are discussed in detail in this section:

WHILE
REPEAT
FOR

## WHILE Statement

Function: Executes a statement or compound statement repeatedly until the condition being tested becomes false.

Format:

```
WHILE <conditional expression> DO
      <statement>
```

Arguments: The conditional expression is any expression that returns a BOOLEAN value. Statement may be either a simple or a compound statement.

Use: The WHILE statement is a well-structured method of repeatedly executing a statement block with a leading decision.

Note: Since the WHILE statement has a leading decision, the statement will not be executed if the conditional expression is false when it is first encountered. Therefore, the condition must have a well-defined value before it is first executed or a run time error will occur.

The condition being tested for must be changed somewhere in the loop. Otherwise, control will never exit the loop and an infinite loop will result.

Examples:

Ex. 1

```
oldval := 100;
newval := 1;
WHILE newval < oldval DO
      newval:=sqr(newval);
WRITELN (newval:10);
```

Ex. 2

```
PROGRAM   gradeavg;
  VAR     score, sum, classavg : REAL;
          total : INTEGER;
          done : BOOLEAN;
  BEGIN
    sum   :=   0;   (* initializing *)
    total :=   0;
    done  :=   FALSE;
    WHILE (NOT done)  DO
      BEGIN
        WRITE ('score:  ');
        READ(score);
        IF score < 0
           THEN done := TRUE
           ELSE
             BEGIN
               sum := sum + score; (* add score *)
               total := total + 1; (* increment total number of scores *)
             END (* ELSE *)
      END; (* WHILE *)
    IF total > 0
      THEN
         BEGIN
           classavg := sum/total; (* calculate the average *)
           WRITELN (' Class average = ', classavg:10:3, ' Student count =',
           total)
         END (* IF *)
  END. (*gradeavg *)
```

REPEAT Statement

Function: Executes a statement or list of statements repeatedly until a desired condition is met.

Format:

```
REPEAT
   |statement|;statement;...||

UNTIL   <conditional expression>
```

Arguments: The statements may be any simple or compound PASCAL statement.

Conditional expression is any Boolean expression that returns a TRUE/FALSE value.

Use: The REPEAT statement is used to execute the list of statements with a trailing decision in a well-structured format.

Note: The statement(s) between the REPEAT and UNTIL reserved words will be executed at least once. This can cause unexpected results if it is not planned for. If a leading decision is desired, refer to the WHILE statement.

The condition that is being tested must be changed somewhere in the statement block; otherwise, control will never exit the REPEAT statement (an infinite loop).

There is no semicolon (;) following the last statement. A semicolon may or may not follow the conditional expression, depending on the succeeding statement, i.e. never before an END or an ELSE.

Examples:

Ex. 1 Sample REPEAT-UNTIL Block of Code

```
IF number > 0
 THEN
   REPEAT
    number: = number + 1; (* increment counter *)
    output: = number*10;  (* calculate value *)
    WRITELN( output )
   UNTIL   output > maximum; (* test *)
```

Ex. 2 Sample Program Using REPEAT-UNTIL

```
PROGRAM testing;
  CONST   maxlength = 50;                      (* the maximum number of
                                               questions *)
  VAR     answersheet : ARRAY[1..maxlength] OF CHAR; (* list of
                                             inputted answers *)
          key : STRING[maxlength];             (* list of correct answers *)
          response: CHAR;                      (* response to a question *)
          totquest, wrong, totquest : INTEGER; (* counters *)

  BEGIN
     wrong := 0; totquest := 0;                (* initializing *)
     key := 'TFFTFTTTFTFFTFTFFTFT.';           (* correct answers *)
     WRITELN;
     REPEAT
        totquest := totquest + 1;              (* increment index *)
        WRITELN('answer number ',totquest);    (* write the question number *)
        READ(response);                        (* read the answer *)
        answersheet[totquest] := response;     (* record answer *)
        IF key[totquest] <> response;
           THEN wrong := wrong + 1             (* keep track of number wrong *)
     UNTIL key[totquest+1] = '.';              (* the end of the test *)
     WRITELN('wrong = ',wrong);                (* output *)
     FOR totquest:=1 TO totquest DO
        WRITELN(answersheet[totquest],'        ',key[totquest])
  END. (* testing *)
```

## FOR Statement

**Function:** Executes a simple or compound statement a predetermined number of times.

**Format:**

```
1)  FOR <control variable>:=<initial value>
        TO <final value> DO <statement>
2)  FOR <control variable>:=<initial value>
        DOWNTO <final value> DO <statement>
```

**Arguments:** The control variable is a user-defined identifier. The initial and final values define the range of values the control variable takes on. The conrol variable and value limits must all be of the same scalar type. They cannot be REAL. The statement may be either simple or compound.

**Use:** The FOR statement is used to execute a simple or a compound statement repeatedly when the number of repetitions is known beforehand instead of being dependent on the results of the loop. Although the same results could be achieved using a WHILE statement, the FOR statement gives the reader more information.

**Note:** The initial and final values are evaluated only once so the limits of the control variable cannot be changed in the loop. After the control variable exits the loop, its value is undefined. Also, its value should never be altered inside of the loop.

The first form of the FOR statement assigns the initial value to the control variable and then increments it by one after each loop. The loop exits when the index value is greater than the final value. If the initial value is larger than the final value the loop will not be executed.

The second form assigns the initial value to the control variable and decrements it by one after each iteration. The loop is exited when the control variable is less than the final value. If the initial value is smaller than the final value, the loop will not be executed.

Examples:

Ex. 1 Example using FOR...TO

```
PROGRAM dates;
TYPE     weekdays = (Sunday, Monday, Tuesday, Wednesday,
                     Thursday, Friday, Saturday);
VAR      days : weekdays;
         date : INTEGER;
BEGIN
  READ(date);
  FOR  days := Sunday TO Saturday DO
       date := date +1;
  WRITELN (next Sunday's date is ', date);
END. (* dates *)
```

Ex. 2 Example using FOR...DOWNTO

```
PROGRAM takeoff;
VAR  countdown : INTEGER;
     error : STRING;
BEGIN
  FOR countdown := 100 DOWNTO 0 DO
   BEGIN
     READLN (error);
     IF error <> 'ON'
       THEN WRITELN(countdown, 'seconds')
       ELSE WRITELN('ERROR!');

   END (* FOR *)
END. (* takeoff *)
```

Ex. 3 Sample program using both forms of the FOR statement.

```
PROGRAM getgrades (input,output);
  (* This program gets grades as input and keeps them in gradeslist.
     It demonstrates the use of both forms of the FOR statement. *)
 CONST  numgrades = 10;
 VAR    grade : REAL;
        gradelist : ARRAY[1..numgrades] OF REAL;
        students  : INTEGER;
 BEGIN
  WRITELN;
  FOR students := 1 TO numgrades DO   (* get the grades *)
   BEGIN
    WRITE('Next grade: ');
    READ(grade);
    gradelist[students] := grade
   END; (* FOR *)
  WRITELN('The grades are :');        (* write the list of grades *)
  FOR students := numgrades DOWNTO 1 DO WRITELN(gradelist[students])
 END. (* getgrades *)
```

## 5.5 CONDITIONAL/UNCONDITIONAL STATEMENTS

It is often necessary to have more than one possible course of action and have the program choose from among them depending on the situation when it is executed. This means that the program must evaluate a condition and select the correct portion of code to execute. In PASCAL this is done through the IF and the CASE conditional statement.

Unconditional transfer of control from one part of a program to another is performed by the GOTO statement.

The following statements are discussed in detail in this section:

IF
CASE
GOTO

## IF Statement

**Function:** Evaluates an expression and chooses between two possible actions.

**Format:**

```
IF <conditional expression>
   THEN <true statement>
  |ELSE <false statement>|
```

**Arguments:** The conditional expression is any expression that returns a BOOLEAN value (true or false). All statements may be simple or compound.

**Use:** When the IF statement is executed, the conditional expression is evaluated. If the result is TRUE, the true statement is executed and control passes to the statement following the IF statement. If the result is FALSE, the false statement is executed and control passes to the statement following the IF statement. If the false statement is omitted then no operation is performed.

The true statement and false statement can be any valid PASCAL statement.

**Note:** It is incorrect to have a semicolon immediately preceding either the THEN or the ELSE.

It is possible to have an IF statement as the statement following either the THEN or the ELSE. This can create confusion as this example shows:

```
IF <conditional expression>
THEN IF <conditional expression>
THEN <statement>
ELSE <statement>
```

Which IF is the ELSE associated with? To clear up the ambiguity, there are two things to remember. First, the ELSE is always associated with the closest IF statement that does not already have an ELSE clause. Second, proper indenting makes it easier to read and makes the nesting levels more obvious. An example of a properly indented nesting of IF statements is:

```
IF <conditional statement>
  THEN IF <conditional statement>
         THEN <statement 1>
         ELSE <statement 2>
  ELSE <statement 3>
```

It is important to remember that the indenting is meaningless to the computer. If the first ELSE clause were removed, PASCAL would associate the ELSE clause containing statement 3 with the nested IF statement, regardless of how the indenting was formatted.

Examples:

Ex. 1

```
IF   ODD(number)
   THEN   oddnum:=oddnum + 1
   ELSE
    BEGIN
      evennum := evennum + 1;
      IF ((number/4) = 1.0 * (number DIV 4))
          THEN div4 := div4 + 1
    END; (* ELSE clause *)
```

Ex. 2

```
IF state <> 0
   THEN   pointer := pointer + 1;
```

Ex. 3

```
PROGRAM order;
 (* THIS PROGRAM READS TWO REAL NUMBERS, PUTS THEM INTO ASCENDING
    ORDER, AND PRINTS THEM OUT.  IT USES AN IF STATEMENT. *)
 CONST  precision = 5;
        field = 10;
 VAR    val1, val2,tempval : REAL;

 BEGIN
  WRITELN;
  WRITE('Give two real values :');
  READ(val1,val2);  (* READ TWO REAL VALUES *)
  IF (val1 > val2)
   THEN
    BEGIN
     tempval := val1; (* SWITCH THE TWO NUMBERS IF *)
     val1 := val2;    (* THEY ARE NOT IN THE CORRECT *)
     val2 :+ tempval  (* ORDER *)
    END; (* IF *)
  WRITELN(val1:field:precision,val2:field:precision)
 END. (* order *)
```

## CASE Statement

**Function:** Transfers control to one of several statement labels depending on the variable's value.

**Format:**

```
CASE  <expression>  OF
||case label|:statement;...|
END;
```

NOTE: Optional groups of elements are underlined to indicate what each contains.

**Arguments:** The expression must evaluate to a user-defined identifier of either scaler or subrange type. The statement associated with the case label that equals the expression is executed. Control then passes to the statement following the CASE statement.

The case label contains one or more constants of the expression type. A case label is equal to the expression if the value of the expression is a constant in the case label.

**Use:** The CASE statement is used when the value of a variable determines which of more than two actions should be taken. It is like a generalized IF statement that is more readable.

**Note:** Each value of the case selector must be represented in exactly one of the label lists. If no action is to be taken, the statement field should be left blank.

There must be at least one value in each case label. Multiple values separated by commas mean that the same action is taken for each of the values. If all the values of a type are not in a case label then the results of executing the CASE statement with the unlisted values is undefined by standard PASCAL. In this case a null statement is assumed.

Examples:

Ex. 1 Example of a CASE Statement with Simple, Compound, and Empty Fields

```
PROGRAM work;
  TYPE    rooms = (livingroom, diningroom, bedroom, kitchen, garage);
  VAR     chores : rooms;
  PROCEDURE vacuum;
    BEGIN
    END;
  PROCEDURE dust;
    BEGIN
    END;
  PROCEDURE settable;
    BEGIN
    END;
  BEGIN
    CASE chores OF
      Livingroom : BEGIN
                     vacuum;
                     dust
                   END;
      diningroom : settable;
      garage, bedroom : ; (* do nothing *)
      kitchen : cook
    END; (* CASE *)
  END. (* work *)
```

Ex. 2 A sample program using a CASE statement

```
PROGRAM writing(input,output);
 CONST  cola = 0.40; fries = 0.45; burg = 0.60; dog = 0.50;
 TYPE   value = 1..4;    (* value is a subrange type *)
 VAR    cost : REAL;
        food : STRING;
        what : value;
        number : INTEGER;
 BEGIN
  cost := 0.0;
  WRITELN;
  REPEAT
   WRITELN('HOW MANY OF 1-COKE,2-FRIES,3-BURGER,4-HOTDOG?');
   READ(number,what);
   CASE what OF
    0 : ;
    1 : BEGIN
         cost := number * cola + cost;  (* calculate the cost *)
         food := 'coke'   (* assign the name of the ordered food *)
         END;
    2 : BEGIN
         cost := number * fries + cost;
         food := 'frenchfries'
         END;
    3 : BEGIN
         cost := number * burg + cost;
         food := 'burger'
         END;
    4 : BEGIN
         cost := number * dog + cost;
         food := 'hotdog'
         END
    END; (* CASE *)
   WRITELN (number, ' ',food)     (* write the order *)
  UNTIL number = 0;
  WRITELN ('TOTAL IS $', cost:6:5)
 END. (* writing *)
```

Ex. 3 A CASE Statement with Simple, Compound, and Empty Fields

```
TYPE    rooms = (livingroom, diningroom, bedroom, kitchen);
VAR     chores : rooms;
            :
CASE    chores    OF
  livingroom : BEGIN
                 vacuum;
                 dust
               END;
  diningroom : settable;
  bedroom : ;
  kitchen : cook
END;  (* CASE *)
            :
```

## GOTO Statement

**Function:** Unconditionally transfers control from one portion of the program to another.

**Format:** GOTO <label>

**Arguments:** The label may be any positive number with one to four digits. All labels must be defined in the label definition part of the program.

**Use:** The GOTO statement is usually used to exit from a loop or in cases of error detection.

**Note:** The GOTO statement is not usually used in structured programming. The readability of a program tends to decline with the increase in GOTO statements. This is because the flow of control is not linear as it is in truly structured programming.

A GOTO statement may jump forward or backward within a level or from an inner to an outer level. However, it may not be used to jump from an outer to an inner level. For example, it may be used to jump from a procedure to its calling program but not the reverse. Another example is that it cannot jump into a WHILE loop but can jump out of one.

Examples:

Ex. 1

```
PROGRAM
LABEL        1,2;
  :
BEGIN
 1: statement;
      :
 WHILE (condition)
   BEGIN
     :
     GOTO 2;
     :
     2: statement 2;
      :
     GOTO 1
   END; (* WHILE *)
      :
END.
```

Ex. 2

```
PROGRAM fakefor (input,output);
 LABEL   100,200;
 VAR     index : 1..100;
         initial,final : INTEGER;
 BEGIN
  WRITELN;
  WRITE('initial and final values: ');
  READ(initial,final);    (* get limits for the FOR *)
  index := initial;
  100 : IF index > final THEN GOTO 200; (* leave loop if
                                 condition is satisfied *)
   WRITELN(index);
   index := index + 1;  (* update counter *)
  GOTO 100;  (* jump to the top of the loop *)
  200 : WRITELN('next statement')
 END. (* fakefor *)
```

1

# SECTION 6
# STANDARD DATA TYPES

# SECTION 6
# STANDARD DATA TYPES

## 6.1 INTRODUCTION

All programs act on data either in the form of variables or constants. The main difference is that a variable's value can change during the execution of the program. Every variable in a program has an associated type which determines the values it can have and the operations that can be performed on it. PASCAL has four standard types: INTEGER, REAL, BOOLEAN, and CHAR. Each of these will be discussed in turn in this section.

## 6.2 INTEGER

The word "integer" is used in the normal mathematical sense: an integer can be any positive or negative whole number. Since all computer representations of numbers must be finite, the maximum representable integer in PASCAL, called maxint, is 32767. The smallest possible number is the negative of maxint. Any variable that is assigned a value outside that range during execution will cause a run-time error. Some examples of integers are: 3, 0, -521.

There are five operators associated with INTEGER types: +, -, *, DIV, and MOD. The first three are the usual addition, subtraction, and multiplication, respectively, used in everyday arithmetic. DIV is the INTEGER divide which divides two integer numbers and then truncates the remainder so that the result is an integer. Special care should be exercised when DIV is used because if the first number is smaller than the second the result will be zero. The operator "/" can also be used for division but the result is a real number rather than an integer. The last operator, MOD, finds the remainder when two integers are divided together. The result will always be an integer. Some examples of these operators are:

| | | | |
|---|---|---|---|
| 5 + 3 = 8 | 4 DIV 2 = 2 | 4/2 = 2.0 | 4 MOD 2 = 0 |
| 5 - 3 = 2 | 7 DIV 6 = 1 | 7/6 = 1.16667 | 7 MOD 6 = 1 |
| 3 - 5 = -2 | 2 DIV 4 = 0 | 2/4 = 0.5 | 2 MOD 4 = 2 |
| 5 * 4 = 20 | 13 DIV 3 = 4 | 13/3 = 4.33333 | 13 MOD 3 = 1 |

The subtraction operator, "-", can be a unary minus and act as the negation sign, i.e. -45.

All the relational operators, <, <=, =, >=, >, <>, can be applied to INTEGER variables. The results of these expressions are always BOOLEAN. Examples:

```
(3 < 5) = True      (3 >= 5) = False     (5 <> 5) = False
```

See Appendix E for a summary description of all operators and their operands.

There are also some important standard functions that give INTEGER results:

| | |
|---|---|
| abs(x) | If x is an INTEGER variable, the outcome will be the absolute value of x. |
| round(x) | x must be a REAL variable, the result is the value x rounded off to the nearest integer. |
| trunc(x) | x is a REAL variable, the result is the whole number part of x. |

None of the above functions assign the computed value to x.

Note: Type INTEGER will reserve two bytes per value.

<u>Examples</u>:

<u>Ex. 1</u>
```
       x = -4.8;
       val:= trunc(x);
       WRITELN(x:10:3,val:10);
output:  -4.80E+00      -4
```

<u>Ex. 2</u>
```
       x = -4.8;
       val:= round(x);
       WRITELN(x:10:3,val:10);
output:  -4.80E+00      -5
```

```
Ex. 3  ix:= -45;
       val:= abs(ix);
       WRITELN(ix:10,val:10);
output:  -45          45
```

## 6.3 REAL

REAL values are rational numbers. PASCAL represents REAL numbers either in fixed point or scientific notation (e.g., 452.39 or 4.5239E+02 respectively).

The E in the second format means "the first value times ten to the power of the second number," i.e.,

$$4.5239E+02 = 4.5239 \times 10^2 = 452.39$$

The computer can only represent a finite number. Hence, all REAL variables, R, must be within the range 2.93874E-39<R<1.70141E+37. If a variable goes outside of this range during the execution of a program, a run-time error will occur.

Another important property of REAL numbers is their precision. Calculations involving REAL values will be correct to six places. This is also the maximum number of digits that can be written out using the formatting described in Section 2.1. It is important to remember that precision errors can accumulate when many calculations are performed and can result in gross errors. These errors must be trapped for by the programmer.

There are four operators that can take REAL variable operands. They are addition, subtraction, multiplication, and division, (+, -, *, /) respectively. All expressions are evaluated from left to right using standard operator precedence, i.e. going from the highest priority level to the lowest:

```
(,)  -  Highest priority
*,/
+,-  -  Lowest priority
```

Since all of the operators can take both INTEGER and REAL operands, it is important to note that an operator that has both a REAL and an INTEGER operand will always produce a REAL result. This is especially important because it is not possible to assign the result of a REAL expression to an INTEGER variable.

All of the relational operators can be used with REAL numbers. However, there is a certain risk involved because of variances in precision. If a variable has gone through many calculations, the accumulated errors may make a theoretically correct expression such as a=b incorrect. Also, two numbers with the seventh significant digit different will be treated as though they were equal. For example, 1000000.0 = 1000000.1 because both are represented as 1.0E+06 in memory. It is more accurate to test equality of REAL variables by abs(a-b)<errorrange where errorrange is the amount of precision that is significant to the problem.

Refer to Appendix E for a summary of the operators, their operands, and the resultant types.

The standard function abs(x) produces the absolute value of x which is REAL if x is REAL. TYPE REAL reserves four bytes per value.

## 6.4 BOOLEAN

BOOLEAN variables may have one of two logical values: true or false. Their primary use is for controlling loop and statement execution.

There are three logical operators that can be applied to BOOLEAN operands.

| | |
|---|---|
| NOT X | logical negation |
| X AND Y | logical conjunction |
| X OR Y | logical disjunction |

They are listed according to their precedence, with NOT always being applied first unless parentheses alter the order. The truth table below shows the result that each operator produces according to the values of its operands. The operands are shown as A1 and A2 and the outcome of each expression, depending on the values of A1 and A2, are read across the table.

| X | Y | X AND Y | X OR Y | Z | NOT Z |
|---|---|---|---|---|---|
| true | true | true | true | true | false |
| true | false | false | true | false | true |
| false | true | false | true | | |
| false | false | false | false | | |

As this shows, both variables must have true values for an AND expression to be true, but only one must be true for an OR to be true.

The following are examples of compound expressions using the logical operators. If the following variables were declared:

```
VAR  big, small, empty, full:  BOOLEAN;
```

then some expressions might be:

```
NOT big OR empty

small AND big OR empty AND full
```

The first expression would be executed as though it had been written:

```
(NOT big) OR empty
```

because NOT is always performed first. The second would be calculated like:

```
(small AND big) OR (empty AND full)
```

because AND has precedence over OR. Had the expression included a NOT, such as

```
small AND NOT big OR NOT empty AND Full
```

it would have produced the same result as

```
(small AND (NOT big)) OR ((NOT empty) AND full)
```

Once again because of NOT's precedence over AND and OR.

There are also seven relational operators that may have any type of scalar operand that has an order such as INTEGER or REAL. They always return BOOLEAN results. They are:

| | |
|---|---|
| < | less than |
| <= | less than or equal to |
| = | equal to |
| >= | greater than or equal to |
| > | greater than |
| <> | not equal to |
| IN | include |

For example, the following are true statements:

```
3 < 5 = true
7 >= 20 = false
```

If relational expressions are used with logical operators, the relational expressions must be surrounded by parentheses. For example, using these declarations,

```
VAR        count, max : INTEGER;
           velocity, distance, miles : REAL;
           next, character: CHAR;
           last:  BOOLEAN;
```

the following expressions could be formed:

    count < max
    (distance < miles * velocity) OR (next = character) AND last

Refer to Appendix E for a summary of the uses, legal operands, and results of the operators. Note that the PASCAL type BOOLEAN is defined so that (false < true).

## 6.5 CHAR

A variable of the type CHAR has a character value. This can be any symbol from the ASCII set (American Standard Code for Information Interchange). A list of the characters and their numberic codes can be found in Appendix F. The ordinal values range from zero to 255.

All CHAR literals are enclosed in apostrophes:

    'A'  represents a letter A
    ' '  represents a blank
    '''' represents a single apostrophe
    '3'  represents the character 3

Every symbol is ordered and has an ordinal value. The only operators that may be used with CHAR variables are the standard relational ones:

    < , <=, =, >=, >, <>

When a relational compare is performed, the operators are actually comparing the ordinal values of the two characters rather than the symbols themselves. For example, 'a' < 'b' = true because the ordinal value of 'a' is smaller than the ordnal value of 'b'.

There are two standard functions that operate on CHAR variables:

| | |
|---|---|
| ord(ch) | Gives the decimal ordinal value of the character ch. The result is an INTEGER value. |

chr(i) Gives the character whose ordinal value is the integer i if $0 \leq i \leq 255$. Any i that is outside of this range or is not an integer will cause either an incorrect response or an execution-time error. The result of the function is of type CHAR.

The ordinal values can be used as character constants. The general form of a character constant is:

```
'<character>'
'(:<INTEGER constant>:)'
```

Since the ordinal value of 'E' is 69, using the declaration:

```
VAR charac : CHAR;
```

the following are equivalent:

```
charac := 'E';
charac := '(:69:)';
```

These ordinal values are the decimal values shown in Appendix F. They may be used anywhere in the program. Some examples using the standard functions and 'E':

```
ord('E') = 69  and chr(69) = 'E'
```

# SECTION 7
# USER DEFINED TYPES

SECTION 7

USER DEFINED TYPES

## 7.1 INTRODUCTION

Many of the concepts basic to PASCAL have already been presented. The beginning programmer knows all that is needed to write simple programs. More advanced programmers will find that this section and those that follow present concepts that can increase a program's sophistication and flexibility. A complete understanding of Sections 2 through 6 will be needed for these sections.

Each of the variable types declared in this section are declared in the TYPE definition part of a program. This was previously described in Section 2.1. The TYPE definitions are placed between the CONST and the VAR declarations in program declarations. The general format is:

```
TYPE  <type identifier> = (<type description>);
      |type identifier = (type description);...|
```

A type describes a template, not actual storage.

After they are defined, the type identifiers are used to define variables in the VAR declaration section. Once a type identifier is defined, it operates the same way that the standard types - REAL, INTEGER, CHAR, and BOOLEAN - do. So the following could be used as part of a program:

```
TYPE         cowfoods = (milk, cheese, meat);
VAR          food : cowfoods;
                  :
```

Now the identifier food can only be assigned one of the three values from the enumerated type list.

## 7.2 SCALAR

A scalar type declaration is the set of constant values that a variable may assume. The general format for the type descriptor is:

```
(<constant>|,...|)
```

The type identifier can be any user-defined identifier. The constant list is the ordered ascending list of values. The list consists of a series of constants separated by commas. The constants are not declared in the CONST declaration field because their values are defined by their order in the list. For example:

```
TYPE days = (Sun, Mon, Tues, Wed, Thurs, Fri, Sat);
```

Days is the type identifier; the enumerated types are the values Sun through Sat. They can be used almost anywhere constants can, i.e.,

```
FOR whatday := Sun TO Sat DO...
```

The order in which the constants are listed defines their ordinance values. Going back to the example above, Sun < Sat and Wed < Thurs.

Therefore, succ, pred, or ord may be used with enumerated types to operate on these constant values.

For example:

```
IF succ (Tues) = Wed then WRITELN ('PAYDAY');
```

### Restrictions on Scalar Constants

There are only a few restrictions on scalar constants:

1) They may not exist in more than one type list.
2) They can be assigned to variables but the variable must be declared the same type as the scalar constant.
3) They cannot be read or written directly.

Scalar variables may be operated on only by the logical operators, <, <=, =, >=, >, <>, which return BOOLEAN values. Both operands in an expression must be of the same type. There are three standard functions designed specifically for scalar type arguments: succ(Y), pred(Y), and ord(Y). They find the element in the list succeeding Y, the preceding value, and the ordinal value of the constant, Y, respectively. Some examples of each are:

```
TYPE   days = (Sun, Mon, Tues, Wed, Thurs, Fri, Sat);
  pred(Tues) = Mon
  succ(Tues) = Wed
   ord(Tues) = 2
   ord(Sun)  = 0
```

There are several things to note about these standard functions. The first element in the constant list has no predecessors and the last element has no successors. Also, the ordinal value of the first element in the list is zero, not one.

<u>Examples</u>:

```
TYPE  meals = (breakfast, coffeebreak, lunch, dinner, snack);
      animals = (dog, cat, hamster, mouse, fish, snake);
      courses = (math, English, biology, philosophy,
                computerscience);

VARS  pet : animals;
      eat : meals;
      homework : courses;

Main Section:
   IF succ(homework) = biology then ...;
   IF ord(homework) = 0 then ...;
   IF pred(eat) = dinner ...;

WHILE (succ(pet) > hamster) DO ... END;
WHILE (pred(eat) < lunch) DO ... END;
```

## 7.3 SUBRANGE

A type may be defined as a subrange of any other defined scalar type, which is called its associated scalar type. It is defined by two constants - a minimum and a maximum value where the minimum must be smaller than the maximum and both must be of the same type. The variable has the same type as the two constants that define it. Subrange variable types may be INTEGER or CHAR but not REAL.

The format for defining a subrange type is:

```
TYPE   <type identifier> = <constant>..<constant>;
       |type identifier = constant..constant;...|
```

Subrange variables may then be declared in the usual way in the variable declaration section:

```
     TYPE   date = 1..31;
     VAR    day : date;
```

A variable can define its type directly using the general form:

```
     VAR    <identifier> : <constant>..<constant>
```

Both of the following produce the same definition for counter, but the first is more explicit and therefore is used more often.

```
     TYPE   index = 1..100;
     VAR    counter : index;
```

and:

```
     VAR    counter : 1..100;
```

Although counter could have been defined as INTEGER type (since both constants are integers) the subrange definition allows range-checking and gives the reader more information. If the range checking switch is set, the compiler will produce code to check all assignments to subrange type variables for values outside of the legal range. The format for the switch option si:

```
(*$R+*)
```

The default value is R+, the switch being on. To turn it off, exchange the plus sign with a minus sign (-) and place the "dollar sign comment" near the top of the program code. See Appendix B for more information.

Any operator that can operate on a scalar type can operate on its subrange variable. Therefore, if variables are defined as the same type but with different subranges, they may be used in the same expression.

Example:

```
VAR   people : 1..300;
      drinks : 1..10000;
      hour,counter : INTEGER;
          :
counter := hour*people*drinks;
```

Subrange variables may be found on both sides of the assignment sign. However, if an assignment is made that is outside of a variable's range and range checking is enabled, a run time error will occur.

Some more examples of subranges are:

```
TYPE   alphabet = 'a'..'z';     (* CHAR subrange *)
       digit = '0' .. '9';      (* CHAR subrange *)
       weekday = (Sunday, Monday, Tuesday, Wednesday, Thursday,
                  Friday, Saturday);
       workweek = Monday .. Friday;
       number = 1 .. 25;        (* INTEGER subrange *)
```

## 7.4 SET

A set is a collection of values that are all of the same type. Any scalar type may be an element in a set provided they are both of the same type.

A set type is defined using the following format:

```
TYPE    <set type identifier> = SET OF <base type identifier>;
```

Although other definitions may be dispersed between these declarations, the order must be as stated. The constant list can be any series of scalar values as described in Section 7.1. All identifiers are user-defined. If the set type is a standard type - INTEGER or CHAR - the base type identifier is not defined and the appropriate reserve word for the type is used to define the set type identifier. For example:

```
TYPE    gradevals = SET OF INTEGER;
VAR     grades : gradevals;
```

completely defines grades if it is INTEGER type, but

```
TYPE    title = (professor, associateprof, assistantprof,
                 lecturer);
        teachers = SET OF title;
VAR     faculty : teachers;
```

is needed to define faculty using the non-standard type.

A set may be described by a list of its values enclosed in square brackets, i.e. [1, 3, 7, 8] might be the members of a set defined by the INTEGER subrange 0..9. If the values are consecutive, only the first and the last elements need to be shown after it is first defined. For example, if the set fruits contains the following elements,

apples, oranges, bananas, strawberries, peaches, pears it can be written

```
     [apples..pears]
```

or a part of it can be used:

```
     [oranges..strawberries] = [oranges, bananas, strawberries]
```

A set may be empty, in which case it is written: [].

There are four operators used exclusively for sets. The first three are:

```
     +   set union
     *   set intersection
     -   set difference
```

The union of two sets forms a set containing the elements from each. For example:

[apples, oranges, peaches] + [strawberries, peaches]

equals:

[apples, oranges, strawberries, peaches]

The intersection of two sets forms a set containing only those elements found in both sets.
For example:

[apples, oranges, peaches] * [strawberries, peaches]

equals:

[peaches]

The difference of two sets is those elements of the first set that are not members of the second set:
For example:

[apples, oranges, peaches] - [strawberries, peaches]

equals:

[apples, oranges]

The relational operators are also used with sets, but their meanings are different. The following operators all return BOOLEAN values.

```
=    set equality
<>   set inequality
<=   is contained in
>=   contains
```

Two sets are equal only if every element in one is in the other. The order does not matter. A set contains another only if every element in the second set is in the first set. Some examples of these operations are:

```
[apples, oranges] = [oranges, apples]
[apples, pears] <> [apples, oranges]
[apples] <= [apples, oranges, pears]
[apples, oranges, pears, peaches] >= [oranges, peaches]
```

Each of these expressions would return a true value.

There is also a reserved word, IN, that tests for set inclusion. It returns a BOOLEAN value by testing if the first value in the expression, a scalar, is in the set described. An example is:

```
[oranges]  IN fruit
```

or:

```
[oranges]  IN [apples, pears, oranges]
```

As fruit was declared earlier, the results are both true.

The assignment sign is used in the usual way when it is used in conjunction with sets. All sets must be assigned to declared set variables.

```
lunchfruit := [apples, oranges, pears, peaches];
dinnerfruit := [strawberries] + lunchfruit;
scratchpaper := [];
```

A set cannot be read in or written out using the READ or WRITE commands.

Some further examples:

```
TYPE   space = (house, apartment, condominium, townhouse, tent,
               trailerhome);
       siblings = (Karen, Cathy, Debbie, Holley, Michael, John,
                  David);
       livingplace = SET OF siblings;
       alphaset = SET OF CHAR;

VAR    family : kids;
       home, setting : livingspace;
       alpha : alphabet;
```

# SECTION 8
# STRUCTURED DATA TYPES

SECTION 8

STRUCTURED DATA TYPES

## 8.1 INTRODUCTION

Section 7 discussed unstructured or simple types while those in this and the next two sections are structured types. Structured types are different from simple types because they are compositions of other types. The types of the components and the structuring methods are what characterize the different structured types.

## 8.2 ARRAY

An array type has a fixed number of ordered components referenced by the same identifier name. The name, the number of components, and the component type are specified when the array is defined. This is done by specifying a base, or component, type and an index type. The component type may be any structured or unstructured type. The index must be either a scalar or a subrange type. It may not be REAL.

The format for an array type is:

```
TYPE  <array identifier>= ARRAY[<scalar>] OF <base type>
```

The array identifier may be any user-defined identifier. The index must be in one of two forms:

1) The identifier name of a defined scalar type:

```
TYPE        numbers = (one, two, three);
            codenames = (owncar, children, spouse, pets);
            matrix = ARRAY[numbers] OF REAL;
            questionnaire = ARRAY[codenames] OF CHAR;
```

2) A subrange of a defined type:

```
TYPE        answers = (Yes, No);
            codenames = (owncar, children, spouse, pets);
            questionnaire = ARRAY[children..pets] OF answers;
            matrix = ARRAY[1..10] OF REAL;
```

The base type may be any legal type. If it is other than a standard type, it must be defined before it appears in the array type definition.

The array type is then used to define variables in the VAR section in the same way that simple types are used.

An element of an array is accessed using the array identifier name followed by the index enclosed in square brackets. An array element may be used anywhere that a simple variable of the same type may be used.

Example:

```
    TYPE        class = (John, Mary, Sue, Karen, Debbie, Mike
                         Dave, Alan);
                classcores = ARRAY[class] OF REALS;
    VAR         grades : classcores;
                who : class;
                  :
            who := Karen;
            grade[who] := 100;
            grade[Alan] := 90;
                  :
```

Since the base type of an array may be any legal type it is possible to have a base type that is a structured type such as an array:

```
    TYPE  matrix = ARRAY[1..10] OF ARRAY[1..5] OF REAL;
```

This defines a two-dimensional array. It is customary to use the following abbreviated form:

```
    TYPE  matrix = ARRAY[1..10,1..5] OF REAL;
```

These two definitions are equivalent, but the second is easier.

The form can be generalized for an array of n-dimensions by:

```
TYPE <ident> = ARRAY[index1,index2,...,indexn]
                                      OF <base type>
```

Each of the indices must be explicitly defined and included in the array definition.

When a two-dimensional array, matrix (i,j), is being referenced, "i" refers to the rows and "j" to the columns. So, matrix (2, 3) is the element in the second row <u>and</u> the third column of the matrix. It would be in the position marked by the X:

| | 1 | 2 | 3 | 4 | |
|---|---|---|---|---|---|
| 1 | | | | | |
| 2 | | | X | | |
| 3 | | | | | |
| | | | | | |

Some multidimensional arrays are:

```
TYPE          square = (' ', 'X', '0');
              board = ARRAY[1..3,1..3] OF square;
              board3d = ARRAY[1..3,1..3,1..3] OF square;
```

"board" is a three by three matrix that may contain blanks, X's and 0's. This might be used to represent a normal tictactoe gameboard. "board3d" represents three matrices like "board" put together. It could be the gameboard for a three dimensional tictactoe game.

There are several types of arrays that warrant special attention. These are:

1) PACKED arrays
2) Arrays with BOOLEAN base types
3) STRINGS

They will each be discussed in turn.

Packed Arrays

Often there are times when the array being manipulated is composed of a type that is not an integral number of words such as characters. The standard ARRAY definition places one character in each word in memory. However, a character representation does not require a full word so the extra space is wasted. This waste could be significant if the array is very large. A packed array minimizes the waste by packing more than one character into each word in memory.

The packed array does have a drawback. The word a character is stored in must be unpacked before the correct character can be accessed. Although the conversion is done automatically, it does require additional processor time.

The relative importance of the minimized waste and conversion time must be determined by analyzing each project separately.

The format for a packed array is:

```
TYPE  <array ident> = PACKED ARRAY[<scalar>] OF <base type>
```

The array identifier, index, and base type are the same as those explained for unpacked arrays.

Arrays with BOOLEAN Base Type

An array with a BOOLEAN base type can be used like a set. Each element has a value of either true or false, which can be used to represent inclusion in the "set". Programs that use BOOLEAN array representation are slower so a set should be used whenever possible. The advantage of the BOOLEAN array representation is that it can be packed so that it can hold more elements in less space. This would be important if the set were very large.

String Arrays

A STRING is one form of a packed array of characters. It is declared using the following format:

```
VAR  <string ident> = STRING[maxlength];
```

As the format shows, the string's maximum length must be declared. This length is stored in the first word of memory that is assigned to the string. Then, if it is assigned a string of characters that is shorter than maxlength, the length value in memory is changed to the new length. However, if it is assigned a string that is longer than maxlength, a compile time error will result. This is true of all arrays.

As with other variables, it must be possible to assign a value to an array element and to assign one array to another. In PASCAL, the assignment symbol (:=) performs these functions. However, there is one restriction: only like types may be assigned to each other. Although an array, a packed array, and a string may all have the base type CHAR, these array types are not represented the same way in memory. Therefore, none of the following would work:

```
TYPE        chr = ARRAY[1..10] OF CHAR;
            pcked = PACKED ARRAY[1..10] OF CHAR;

VAR         charray : chr;
            pckdarray : pcked;
            str : STRING[10];
            index : INTEGER;
```

```
charray := str;              FEL
pckdarray := str;            FEL
charray := pckdarray;        FEL
str := pckdarray;            FEL
pckdarray := charray;        FEL
str := charray;              FEL
```

However, since they all have base type CHAR, each of the following would compile and run if the same declarations were used:

```
charray[index] := str[index];
pckdarray[index] := str[index];
charray[index] := pckdarray[index];
str[index] := pckdarray[index];
pckdarray[index] := charray[index];
str[index] := charray[index];
```

Each assignment statement is valid because a CHAR variable (str[index], char[index], or pckdarray[index]) is being assigned a character value.

There are also restrictions on the array types that may be read in or written out. The non-standard scalar and the BOOLEAN array types cannot have I/O performed on them at all. PACKED arrays of characters can only be written out because of the problems that conversions entail. Standard scalar type arrays can only be read in or written out one character at a time, i.e. using an index. A string can be read in or written out one character at a time or all at once. Table 8-1 summarizes these results.

Table 8-1. Restrictions on I/O with Arrays

| | ARRAYS | | | PACKED ARRAYS | | STRINGS |
|---|---|---|---|---|---|---|
| | INTEGER/ REAL | character | non-standard scalar | character | BOOLEAN | |
| READ | I | I | N | N | N | DI |
| READLN | I | I | N | N | N | DI |
| WRITE | I | I | N | I | N | DI |
| WRITELN | I | I | N | I | N | DI |

Key

I - Only using an index, i.e. 1 character at a time.

N - Not at all.

DI - Done directly.

## 8.3 RECORD

A record is a type with a user-defined structure that incorporates several components, each of which have distinct properties. The different components are called fields and are accessed by name rather than with an index.

The record is defined as a variable type. The format is:

```
TYPE        <record ident>=
               RECORD
                  <component identifier>|,...| :<base type>
                  |component identifiers : base type;|
                  |CASE section|
               END;
```

The record identifier is any user-defined variable name. The component definitions must be enclosed by the reserved words RECORD and END; the identifiers are user-defined and separated by commas. The base types may be any structured or simple type, but must be defined before they appear in the record definition. A semicolon (;)

separates the components, but one does not appear between the last component's base type and the reserved word END.

A field in a record is accessed using the record identifier and the component identifier separated by a period and in that order. The general format is:

<record ident>.<component ident>

The base type of a component may be a record so it is possible to have nested records.

Example 1:

```
TYPE        pername =
               RECORD
                  first : STRING[10]
                  midinitial : CHAR;
                  last : STRING[15]
               END;
            person =
               RECORD
                  name : pername;
                  address : STRING[25]
               END;
VAR         people : person;
```

Example 2:

```
TYPE        person =
               RECORD
                  name:  RECORD
                            first : STRING[10];
                            midinitial : CHAR;
                            last : STRING[15]
                         END;
                  address : STRING[25]
               END;
VAR         people : person;
```

In both cases, the last name of the person would be accessed by:

```
people.name.last
```

This is an obvious extension of the general form displayed above. Accessing a field is even easier using the WITH statement. This statement is defined in the usual format on the next page.

Packed Records

It is also possible to define PACKED RECORDS. If one or more of the record's fields has a base type with an internal representation that does not require an integral number of words, such as character type, then the field can be packed into less space in memory. Although this saves space, a conversion is required each time the field is used.

Example:

```
TYPE    rec  = RECORD
                 A1:0..255;
                 A2:CHAR
               END;
        prec = PACKED RECORD
                 A1:0..255;
                 A2:CHAR
               END;

VAR     normspace : rec;
        savespace : prec;
```

Note that savespace will occupy 2 bytes in memory while normspace needs 14 bytes.

Because of the way in which a field is accessed, each fieldname within a record must be unique. However, a record's fieldname may be the same as a variable or type identifier outside the record because the record identifier differentiates them.

## WITH Statement

**Function:** References a record once for one or more field accesses.

**Format:** WITH <record ident>|,record ident,...| DO
<statement>

**Argument:** The record identifier is any defined record name. There may be more than one record identifier separated by commas in a WITH statement. The statement can be any simple or compound statement.

**Use:** The WITH statement increases the efficiency with which the same component of a record or different components of the same record can be accessed repeatedly. Within the statement section the components can be refered to by only the field names.

**Note:** The WITH statement locates the record or records involved. Then, all field references within the statement are made directly - the record is not relocated for each one. This greatly increases the program's efficiency when there are multiple accesses.

Since the compiler assumes that a reference in the statement part of a WITH statement is to the specified record, the record identifier is not necessary in field references. Used properly, this can save both the programmer and the computer a great deal of time.

It can also create confusion if there is a variable name that is the same as a field name in the specified record. If so, the variable cannot be referenced inside a WITH statement involving the record because the record's field will be assumed instead. Therefore, great care must be exercised if duplicate identifiers are used in conjunction with WITH statements.

Example:

```
     TYPE         money =
                    RECORD
                      quarter, dime, nickel, penny : INTEGER
                    END;
     VAR          change : money;
                  total : REAL;

     BEGIN
       WITH change DO
         total := quarter * 0.25 +
                  dime * 0.10 +
                  nickel * 0.05 +
                  penny  * 0.01;
     END
```

Record Assignment

A record, as a structure, cannot be used as an operand for any operator. This is because there is no ordering associated with records and because the operator may not be compatible with all the record's fields. However, a field in a record may be used with any operator that is compatible with its base type.

A record cannot be assigned a value because of the ambiguity with the fields and their types. If two records are declared exactly the same type then one may be assigned to the other using the assignment symbol (:=). For example:

```
TYPE        days = 1..31;
            mo = 1..12;
            date =
               RECORD
                  day : days;
                  month : mo;
                  year : INTEGER
               END;

VAR         issuedate,todaysdate, expirationdate : date;
```

This:

```
issuedate := todaysdate;
```

is equivalent to the sequence:

```
issuedate.day := todaysdate.day;
issuedate.month := todaysdate.month;
issuedate.year := todaysdate.year;
```

A field can be assigned a value using the assignment symbol if the value and the base type are compatible:

```
issuedate.day := 10;
```

A record may be passed in the parameter list of a function or a procedure, but it may not be used as the return value of a function because it does not represent a simple type.

Illustrated Example - Arrays, Records and WITH Statements

This program uses records, nested records, WITH statements, and identical variable and field names.

```
PROGRAM records;
 TYPE   marry = (single, married, divorced, widowed);
        money =        (* record type for INCOME in PERSON *)
         RECORD
          salary, other : REAL
         END;
        person =       (* record of the customers' data *)
         RECORD
          name :  PACKED ARRAY[1..25] OF CHAR;
          addr :  PACKED ARRAY[1..30] OF CHAR;
          marstatus :  marry;
          dependents :  INTEGER;
          income :  money    (* a record type *)
         END;
 VAR   customer :  ARRAY[1..10] OF person; (* records of all customers *)
       scratch :  person;   (* a workarea for the input  *)
       name :  STRING[25]; (* used for string input *)
       addr :  STRING[30];
       marstat,ans :  CHAR;
       I,J :  INTEGER;
       more :  BOOLEAN;
BEGIN
 WRITELN;
 J := 0;  more := true;     (* initializing *)
```

```
 WHILE more = true DO      (* beginning of loop *)
  BEGIN
   J := J + 1;       (* increment index *)
   WRITELN('Customer''s name?');
   READLN(name);
   FOR I := 1 TO 25 DO         (* assign the name *)
    scratch.name[I] := name[I];
   WRITELN('Address?');
   READLN(addr);
   FOR I := 1 TO 30 DO        (* assign address *)
    scratch.addr[I] := addr [I];
   WRITELN('Marital status?  (M,S,D,W)');
   READLN(marstat);
   WITH scratch DO
    CASE marstat OF    (* assign the marital status *)
     'M' :  marstatus := married;
     'S' :  marstatus := single;
     'D' :  marstatus := divorced;
     'W' :  marstatus := widowed
     END; (* CASE *)
  WRITELN('Number of dependants?');
  READLN(scratch.dependents);
  WRITELN('Salary and other income, in that order?');
  WITH scratch, income DO
   READLN(salary,other);
  customer[J] := scratch;   (* saving the workarea *)
  WRITELN('Are there more customers?');
  READ(ans);
  IF (ans <> 'Y') AND (ans <> 'y') OR (J = 10) THEN more := false
 END  (* WHILE *)
END. (* records *)
```

Record Variants

Records declared to be the same type may sometimes vary in the number and types of their components. This is done in the variant part of the record declaration.

A record may contain a fixed part, a variant part, or both. If a record contains both, the fixed part must come first.

The variant part is superficially like a case statement. It is of the form:

```
RECORD
  |fixed part|
  CASE |tag field:|<type ident> OF
    <case element>|;...|
    <case label list>
END
```

Where <case element> is:

<case lable list: (|field identifiers : field type |)

(Note: The above line may be repeated for as many case lists as necessary.)

The tag field is an identifier that is defined by the type identifier which must be a scalar type. It can be defined in the fixed part of the record. The case label lists are the values of the type identifier. The field identifiers are defined by the field types, which can be any structured or simple type. Associated with one field type there may be multiple case labels and field identifiers, with commas acting as delimiters in each list. An example of a variant part of a record is:

```
TYPE        kind = (trout, catfish, goldfish, bluegill, salmon);
            animals = (cats, dogs, fish, sheep, cows, pigs);
            animate =
              RECORD
                CASE  tag : animals  OF
                  cats, dogs : (pets, inside : BOOLEAN);
                  sheep, cows, pigs : (food : INTEGER);
                  fish : (both : kind)
              END;
```

The reserved word END is associated with the record declaration, not the CASE statement. However, it operates as the terminator for both since the variant part must appear last in the record declaration.

A field identifier may not be used more than once inside a record, regardless of whether it is in the variant or fixed part. However, it may replicate an identifier of a variable or type that is defined outside the record.

A tag type identifier value is not required to appear in one of the case label lists; however, it is recommended that all values be represented for program security. If no action is associated with a label, the field identifier and type are left blank. For example:

```
pigs : ();
```

There are two ways that the records in the variant part may be nested. The field type may be a record, which is the natural nesting:

```
TYPE         color = (red, white, blue);
             play = (fireengine, pail, ball);
             material = (plastic, wood, metal, cloth);
             test =
               RECORD
                 CASE picture : color OF
                   red : (toys : RECORD
                                   number : INTEGER;
                                   CASE toys : play OF
                                     fireengine, pail, ball:material
                                 END);
                   white :  (nothing : REAL)
               END
```

The nesting can also occur by replacing the field identifiers and type with either a fixed or a variant part of a record. The formats are: fixed part:

```
<case label list> : (<component idents> : <base type>;
                    |component idents : base type;...|)
```

variant part:

```
    <case label list> : (CASE | tag field:|<type field> OF
          <case label list> : (|field idents : field type|);...|)
```

If the case label and field identifier parts have more than one element, they are separated by commas. The formats are the same as for non-nested fixed and variant parts of records. Note that the reserved word RECORD does not appear in either case.

The following example demonstrates the nesting described above and will be used in the discussion on accessing.

```
     TYPE         kind = (animal, plant);
                  sort = (mammal, reptile);
                  life = (caged, free);
                  outside = (dog, cat);
                  inside = (mouse, gerbil, guineapig, ferret);
                  locomotion = (slither, crawl);
                  plantinside = (nonflowering, flowering);
                  form = (tree, bus, flower);
                  creature =
                    RECORD
                      CASE kingdom:kind OF
                        animal : (CASE phylum:sort OF
                                mammal : (CASE care:life OF
                                        free : (species : outside);
                                        caged : (rodent :  inside));
                                reptile : (snake, lizard : locomotion));
                        plant : (unprotected : form;
                               protected : plantinside)
                    END;

     VAR          pet : creature;
```

Note that there are three levels of variant parts and that "plant" labels a fixed part of a record.

The purpose of the variant part of the record is to allow enough flexibility that the record's construction can be developed as the program is executed. This is done by setting the tab identifiers equal to the label of the part of the record that is needed. As an example to illustrate this, assume a program using the definitions above needs to deal with the scalar "mouse". The following sequence of assignment statements would make it possible:

```
pet.kingdom := animal;
pet.phylum := mammal;
pet.care := caged;
pet.rodent := mouse;
```

The general form is:

```
<variable ident>.<tab ident> := <case label>
```

The elements of rodent can now be accessed by pet.rodent throughout the program. If, as the program progresses, a different element is needed, say "flowering" plant, the same type of sequence would be needed, i.e.

```
pet.kingdom := plant;
pet.protected := flowering;
```

However, once the record has been redefined, the earlier values may be lost. If pet.rodent were accessed now, the value might be incorrect. Because of this ambiguity, the record should be defined once and all important values assigned to variables if a new form of the structure is needed.

Variant Record Declarations

Variant record declarations are useful for selecting between various types. For convenience, the following is a list of interchangable types:

| Declaration | Space Occupied |
|---|---|
| ARRAY[1..X] OF INTEGER; | 2 x bytes, each integer stores as low, high byte. |
| PACKED ARRAY[1..X] OF 0..255; | X bytes, each element holding a value 0-255. |
| PACKED ARRAY[1..X] OF CHAR | X bytes, each element holding a character. |
| PACKED ARRAY[1..X] OF BOOLEAN | X bits, each bit loading the value TRUE or FALSE. |
| STRING[X] | X+1 bytes, each element holding a character. STRING[0] holds the dynamic length of the string. |
| STRING | The declaration STRING is equal to STRING[80]. |
| CHAR | One byte holding a character. |
| INTEGER | Two bytes in low, high order. |
| REAL | Four bytes, the exponent and mantissa will occupy bytes 1-3. |

Note that each variable is assigned to the next 16-bit word boundary if the variable does not fit into the current word.

Example:

```
TYPE
  BUFF=PACKED RECORD
    A:0..255;
    B:0..255;
    C:INTEGER;
  END;
```

"A" occupies the first byte, "B" the second byte and "C" byte 3 and 4.

```
PACKED RECORD
  A:0..255;
  C:INTEGER;
END;
```

Here variable "A" occupies the first byte. The integer C occupies byte 3 and 4 since the variable is too large to fit into the last byte of the first word.

Note that in variant records, the amount of storage space allocated to the variant record will be the size of the largest variant among the cases.

# SECTION 9
# POINTER DATA TYPES

## SECTION 9
## POINTER DATA TYPES

### 9.1 INTRODUCTION

Most variables are static, which means that they are allocated during the execution of the procedure to which they are local. In contrast to this, dynamically allocated variables can be created and destroyed as needed. This allows space to be allocated as needed. It also allows functions to return space as results.

Dynamic variables do not occur in an explicit variable declaration so they do not have an identifier by which they can be referred. Instead, they are accessed through a pointer variable that is used to generate the dynamic variable.

A pointer type designates a very small amount of space (usually two bytes) which is used to point to an object. It does not make space for the object. Since a pointer can point to an object, it can be accessed via the pointer once the object has been allocated. The advantage of this is twofold. First, it is possible to switch a pointer variable from one object to another very quickly. Second, it is possible for one object to be referenced by more than one path. This is important because it is a prerequisite for using linked lists.

It is possible to generate new objects from an area called the heap. This generation and manipulation is covered later.

### 9.2 FORMAT

A pointer is declared by preceding the type with a caret. The format is:

```
TYPE     <pointer ident> = ^ <record ident>;
```

Example:

```
TYPE    pointer = ^ class;
        class = RECORD
             link : pointer;
             data : STRING[25]
            END;
VAR     nextstudent,firststudent,laststudent : pointer;
```

Note that the record does not define a variable so it cannot be directly accessed. However, any of the identifiers with the pointer identifier as a type can indirectly access the record.

Pointer Type Components

The pointer identifiers can be used to refer to the data they point to and the pointer itself. Both can be assigned values using the assignment statement. Assume the two variables, next and base, are declared pointer types and are pointing to different records of the same type. The statement, base := next, means that base now points to the same record that next does. The statement "base ^ := next ^ " means copy the value in the record "base" points to into the record "next" points to. These differences in the assignments must be remembered.

Values within a record pointed to by ident may be assigned by:

```
<ident> ^ .<field> := <expression>
```

The field may be of any type and the expression may be anything that returns a value that is valid for the field. For example, if the record's data field is INTEGER:

```
base ^ .data := 10;
base ^ .link := next;    (* link and next are pointer types *)
```

It is certainly possible for a pointer to point to an empty list (i.e. at the beginning of a program). The reserved word NIL represents this case. It may not be used in an arithmetic expression. However, it operates as a value and can be assigned or compared just as any other value can. For example, assign base equal to NIL:

```
base := NIL
```

Lastly, as a list is being built, the new components must be allocated. The predeclared procedure, NEW, does this. The format is:

```
PROCEDURE NEW(<ident>:POINTER TYPE)
```

The identifier must be declared a pointer. When the statement is executed, it points to the location of the new component. Hence,

```
NEW(base);
```

means that base ^ can be used to access the newly allocated record.

The NEW procedure allocates space from an area in core called the heap. This is a dynamic area which shares space with the program stack (where global and local function and procedure variables are allocated). This area can be viewed as a linear array with the stack at one end, the heap at the other, and free space in the middle.

As a program executes and calls procedures, the stack grows and uses some of the free space. As the procedures complete, they automatically return the space and the stack contracts. When the program executes the NEW procedure, the heap expands and the pointer in the NEW statement points to the space just allocated to the heap. The amount of space allocated is a function of the type that the pointer is to point to. Assigning a new value to the pointer via another NEW or assignment statement does not free the space - it must be returned explicitly through the RELEASE or DISPOSE procedures.

It is the programmer's responsibility to insure that the pointer data is managed correctly. Also, the pointers initially contain a garbage value and should be initialized by NEW or with NIL.

There are many applications that use dynamically allocated variables. With one link field, stacks, queues, and rings can be formed. With two link fields in each record, doubly linked lists and rings, and trees can be formed. The following example generates a queue with a First-In, First-Out (FIFO) structure:

Example:

```
PROGRAM Queue(Input,Output);
 TYPE   groceries = STRING[10];
        pointer =^list;  (* pointer to record *)
        list = RECORD
          link : pointer;
          food : groceries
          END;
 VAR    next,front,rear : pointer; (* pointers *)

 BEGIN
  WRITELN;
  rear := NIL; front := NIL;  (* initialize pointers *)
  WRITELN('What is on the list?');
  REPEAT
   NEW(next);  (* create next record *)
   IF (front = NIL)  (* assign links *)
    THEN front := next
    ELSE rear ^ .link :=next;  (* link it onto the end *)
   rear := next; (* assign rear pointer *)
   READLN(rear ^ .food)
  UNTIL (rear ^ .food = '');
  WHILE (rear <> front) DO   (* if rear = front, the whole *)
   BEGIN                     (* list has been written out *)
    WRITELN (front ^.food);
    front := front ^.link (* update link *)
   END  (* WHILE *)
END. (* Queue *)
```

# SECTION 10
# FILE DATA TYPES

1

## SECTION 10
## FILE DATA TYPES

### 10.1 INTRODUCTION

Files are important variable types because they allow large quantities of data to be accessed and retained in secondary memory. Because of this, large data bases can be conveniently stored and easily manipulated. Also, programs that are larger than main memory may be left in files with only those sections being processed residing in memory.

A file is a sequential collection of values that are all of the same type. It is analogous to a tape in that all data is represented sequentially and only one component of a file can be accessed at any one time. A natural ordering is defined through the sequence.

A file is a unique variable type, partly because it is sequential, and, more importantly, because it may exist before and after a program is executed.

There are two standard files that represent the I/O media: the input and output files. They are the default values in most places where a filename is necessary, notably the READ, READLN, WRITE and WRITELN statements.

### 10.2 REFERENCING FILES IN A PROGRAM

All names of files that are referenced in a program must be listed in the program heading. The format is:

```
PROGRAM   <ident>|(fd,...)|;
```

The standard files, input and output, should be included in the variable filename list if READ, READLN, WRITE, WRITELN, EOF, or EOLN is used without a filename. If a filename does not appear in the list but is used in the program, then it is flagged as a local file and as such becomes undefined after the program is completed. The filename will remain in the directory listing, but its contents are undefined and cannot be displayed.

Declaration Format

A file's declaration format is:

```
    TYPE   <file ident> = FILE OF <type>;
```

The file identifier is user-defined. The type may be of any standard or nonstandard type. Note that the number of components is not fixed by the definition. A global variable, the file identifier, must be declared for each file that is referenced in the program.

## 10.3 FILE TYPES

There are four predefined file types in PASCAL: textfiles, record files, physical files and ISAM files. Their declaration formats are:

```
        <file ident> = FILE OF <type>;          Record file
        <file ident> = TEXT;                    Text File
        <file ident> = FILE;                    Physical File
        <file ident> = ISAMFILE;                ISAM File
```

The RESET or REWRITE statement is used to connect the actual file name with the file identifier in the program.

All the files are sequential but the record lengths differ. Files of user-defined types have a fixed record length that is defined by the type. The GET, PUT, SEEK and EOF I/O-statements are used in conjunction with these files.

A TEXT file is implicitly defined FILE OF CHAR but it has a variable record length because it is subdivided into lines. READ, WRITE, READLN, WRITELN, EOF, EOLN, GET, and PUT are the I/O-statements that are available to access it.

A physical file is a special case of the record file. The record length is 256 bytes. Several consecutive records may be read or written using BLOCKREAD and BLOCKWRITE; refer to Appendix G (programs Byteshape, Bytetest and Anbyte) for examples.

The I/O statements used in conjunction with ISAM files are discussed in Section 14.

## 10.4 PASCAL INTRINSICS FOR FILES

Certain PASCAL intrinsics apply to files. The definition and function of these intrinsics are summarized below. A complete description including the format of each can be found in Section 12.

1. Definition: PROCEDURE RESET(<fd>:File|,title:STRING|)

   Function: Positions the pointer to the first element in the file and prepares it for input. "title" is a string of the form: '<fd>'. If the title is included, RESET opens for an existing but previously closed file so that it can be read. In this case the pointer is pointing to the first record. Without the title, RESET moves the pointer to the beginning of the file and reads the first record for the user. Here, the pointer moves to the second record. If the file is not open, it returns an error through IORESULT and the file remains closed.

   Example: (Title is a string and DATA is the volume name.)

   ```
   title := 'DATA:testfile';
   RESET(testfile,title); (* opens file + points to
                             first record *)
   ```

2. Definition: PROCEDURE REWRITE(<fd>:text FILE,<title>:STRING)

   Function: Creates a new file on disk and opens the file. Filename and title are of the same format as they were for RESET:

   Example:

   ```
   title := 'DATA:testfile';
   REWRITE(testfile,title)
   ```

3. Definition: PROCEDURE READ(<fd>:text FILE|,variable list|)

   Function: Reads the next value or values from the file. It can only be used with TEXT files. If the variable is a string, it will read up to the end-of-line character.

   Example:

   ```
   READ(testfile,val1,val2)
   ```

4. **Definition:** PROCEDURE READLN(<fd>:text file|,variable list|)

   **Function:** Reads through the first character on the following line of the file. It can only be used with TEXT files.

   **Example:** READLN(testfile,val1,val2)

5. **Definition:** PROCEDURE WRITE(<fd>:text file|,item list|)

   **Function:** Writes the value(s) to the file. It can only apply to TEXT files.

   **Example:** WRITE(testfile,val1,val2)

6. **Definition:** PROCEDURE WRITELN(<fd>:text file|,item list|)

   **Function:** Writes the value(s) to the file and then inserts a carriage return character. It is only used in conjunction with TEXT files.

7. **Definition:** PROCEDURE GET(<fd>:file)

   **Function:** Reads the next record from the file into a file buffer associated with that file. This buffer can be accessed via a pointer variable whose name is the same as the filename. The file buffer should be assigned the value of the file pointer variable before the GET is done (see example below). The following are equivalent if the file is text.

   **Example:**

| | |
|---|---|
| READ(textfile,value) | value := textfile ^;<br>GET(textfile) |

8. Definition: PROCEDURE PUT(<fd>:file)

Function: Places the value in the buffer variable into the next available position in the file and updates the pointer. If the file is text, the following are equivalent:

Example:

```
WRITE(testfile,value)        testfile ^ :=value;
                             PUT(testfile)
```

9. Definition: FUNCTION EOF(<fd>:file):BOOLEAN

Function: Returns a Boolean value which represents whether or not the end of the file has been reached.

Example:

```
IF EOF(file1) THEN WRITELN ('END OF FILE1');
```

10. Definition: FUNCTION EOLN(<fd>:file):BOOLEAN

Function: Determines whether or not a carriage return character has been encountered in a text file.

## 10.5 CREATING AND USING FILES

The first time that a new filename is encountered in a REWRITE statement, the new file is created and data may be stored in it. However, if the same program is executed a second time, the file already exists so the REWRITE statement returns as error through IORESULT and the file is not opened. The following sequence will create a new file or, if it already exists, destroy and recreate the file:

```
title := 'test:sample';
REWRITE(sample,title);
IF (IORESULT <> 0) THEN
  BEGIN
    RESET(sample,title);
    CLOSE(sample,PURGE);
    REWRITE(sample,title)
  END;
```

The user should provide the identifiers whose fields are encased in brackets. If this sequence is followed, the program will execute without the file being opened.

If a file exists and contains information needed by a program, RESET(<fd>,<title>) should be used to open the file, set the pointer to the first position, and prepare it to be read from.

There are two more advanced forms of files: external and segmented. These are explained in Section 13.

### Examining File Contents

It is often necessary to examine the file once the information has been placed in it. The following sequence will display the file on the screen one sector at a time:

```
-DISKDUMP
 IN <fd>
 DUH 0
```

The file descriptor <fd> is the name of the file in the format shown in Section 1.2. "DUH 0" displays the sectors in hexadecimal starting with Sector 0. The next sector can be displayed by depressing the RET key. Type "END" to exit DISKDUMP. The utility cannot be called from a PASCAL program.

A file appears as one or more sectors that consist of sixteen by sixteen matrices of bytes. One byte is represented by one pair of numbers in a row. Each number or unpacked character is represented in exactly one byte.

Illustrated Examples

The following programs show how files are created and used.

Ex. 1

This program creates and loads a file with input entered at the console.

```
PROGRAM writeletter (Input,Output,letter);
 TYPE   writing = TEXT;
 VAR    letter : writing;         (* The TEXT file *)
        name : STRING[20];        (* the title variable *)
        nextline : STRING[80];    (* the input string *)
 BEGIN
  name := 'DATA:letter';          (* assign the filename *)
  REWRITE(letter,name);           (* create & open the file *)
  IF (IORESULT <> 0) THEN
   BEGIN
    RESET(letter,name);
    CLOSE(letter,PURGE);
    REWRITE(letter,name)
   END;
  READLN(Input,nextline):         (* receive the next line *)
  WHILE(nextline <> 'end') DO     (* test for end condition *)
   BEGIN
    WRITELN(letter,nextline);     (* write line to file *)
    READLN(Input,nextline)
   END (* WHILE *)
 END. (* writeletter *)
```

Ex. 2

Program GRADEAVG creates a file of grade scores and then uses the data in this file for various computations.

```
PROGRAM gradeavg (input,output,scores);
 TYPE   list = FILE OF INTEGER;
 VAR    sum,classavg : REAL;
        scores : list;  (* file of scores *)
        total,score : INTEGER;
        title : STRING[20];
 BEGIN
  title := 'DATA:scores'; (* name of file *)
  REWRITE(scores,title);  (* open new file *)  (* CREATE NEW FILE *)
  IF (IORESULT <> 0) THEN (* close & open file if it existed*)
   BEGIN
    RESET(scores,title);
    CLOSE(scores,PURGE);
    REWRITE(scores,title)
   END;   (* IF *)
  READ(input,score); (* first score *)
  WHILE (score >= 0) DO
   BEGIN
    scores^:= score; (* put the score in file *)
    PUT(scores);
    READ(input,score)
   END; (* WHILE *)
  sum := 0; (* initializing *)
  total := 0;
  RESET(scores);  (* moves pointer to beginning of file and reads 1st
                   score *)
                  (* then pointer points to second record *)
  WHILE NOT EOF(scores) DO (* continue until end of file *)
   BEGIN
    score := scores ^ ;  (* assigns the value *)
    GET(scores);         (* reads next record *)
    sum := sum + score;  (* add score *)
    total := total + 1   (* add to number of scores *)
   END; (* WHILE *)
  classavg := sum/total;  (* calculate the average *)
  WRITELN('Class average =',classavg:10:3,' Student count =',total)
 END. (* gradeavg *)
```

<u>Ex. 3</u>

Program FILETEST displays integers and then stores the integers in a new file.

```
type
  inarr=array[1..10] of integer;
var
  starr:inarr;
  ix:integer;
  outfile:file of integer;
begin
  rewrite(outfile,'numfl');
  if(IORESULT <> 0) then
    begin
      reset (outfile,'numfl');
      close (outfile, PURGE);
      rewrite (outfile,'numfl')
    end;
  for ix:=1 to 10 do
    begin
      writeln('please input an integer value then press CR ');
      read(starr[ix])
    end;
   for ix:=1 to 10 do
     begin
       writeln('integer value= ');
       writeln(starr[ix])
    end;
   for ix:=1 to 10 do
    begin
     outfile:=starr[ix];
     put(outfile)
    end;
   writeln('done');
   close(outfile)
 end.
```

Ex. 4

Program FILENAMES builds a customer complaint file by prompting for name, number, address, complaint and comment data.

```
program FILENAMES;
    type
         ptl= persons;
         pointer='a'..'z';
         person=record
              name,ssnum,address: string[10];
              comment:string
         end; (* person *)
    var
         p:array['a'..'z'] of ptl;
         gang:file of person;
         index:pointer;
         beginix,endix : char;
    begin
      REWRITE(GANG,'DATA:ABCD');
      if (ioresult <> 0) then
           begin
            reset(gang,'data:abcd');
            close(gang,purge);
            rewrite(gang,'data:abcd')
           end;
      beginix:='a';
      endix:='c';
      for index:=beginix to endix do
              begin
                   new(p[index]);
                   with p[index]  do
                   begin
                   writeln('cust. name ? ');
                   readln(name);
                   writeln('ssnumber ? ');
                   readln(ssnum);
                   writeln('address ? ');
                   readln(address);
                   writeln('complaint ? ');
                   readln(comment);
                   end;
                   gang :=p[index] ;
                   put(gang)
              end;
      close(gang)
    end.
```

Ex. 5

Program TEST6 stores an array of integers on disk. It zeros all elements in the array, GETS the integers from the file, and reloads the array. Finally, the array values are displayed on the console. A DISKDUMP follows the program listing.

```
PROGRAM TEST6 (SAMPLE);
 VAR
  TITLE : STRING[9];
  SAMPLE :FILE OF INTEGER;
  J,IX,I : INTEGER;
  INSAM:ARRAY[1..100] OF INTEGER;
 BEGIN
  TITLE:='SEAL:sample';
  REWRITE(sample,title);
  IF (IORESULT <> 0) THEN
   BEGIN
    RESET(sample,title);
    CLOSE(sample,PURGE);
    REWRITE(sample,title)
   END;
  J:=200;
  FOR I:=1 TO 40 DO
   BEGIN
    J:=J+1;
    INSAM[I]:=J;
    SAMPLE^:=INSAM[I];
    PUT(SAMPLE)
   END; (* FOR *)
  CLOSE(SAMPLE);
  FOR I:= 1 TO 50 DO
   INSAM[I]:=0;
  I:=1;
  RESET(SAMPLE,TITLE);
  GET(SAMPLE);
  WHILE NOT EOF(SAMPLE) DO
   BEGIN
    INSAM[I]:=SAMPLE^;
    GET(SAMPLE);
    I:=I+1
   END; (* WHILE *)
  FOR IX:=1 TO 50 DO
   WRITELN(INSAM[IX]);
  CLOSE (SAMPLE)
END. (* TEST6 *)
```

```
-DISKDUMP
00.00.00 D I S K D U M P  R3.01
IN SEAL:XXXX
DUH 0

   S E C T O R  # :        0.

C9 00 CA 00 CB 00 CC 00 CD 00 CE 00 CF 00 D0 00 I.J.K.L.M.N.O.P.
D1 00 D2 00 D3 00 D4 00 D5 00 D6 00 D7 00 D8 00 Q.R.S.T.U.V.W.X.
D9 00 DA 00 DB 00 DC 00 DD 00 DE 00 DF 00 E0 00 Y.Z.[. .]. ._.'.
E1 00 E2 00 E3 00 E4 00 E5 00 E6 00 E7 00 E8 00 a.b.c.d.e.f.g.h.
E9 00 EA 00 EB 00 EC 00 ED 00 EE 00 EF 00 F0 00 i.j.k.l.m.n.o.p.
29 00 2A 00 2B 00 2C 00 2D 00 2E 00 2F 00 30 00 ).*.+.,.-.../.0.
31 00 32 00 00 00 00 00 00 00 00 00 00 00 00 00 1.2.............
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00 00 00 00 00 00 00 00 54 45 53 54 20 20 20 20 ........TEST
4F 53 2E 38 20 50 61 73 63 61 6C 20 33 2E 30 31 MS.8 Pascal 3.01
20 20 20 46 69 6C 65 3A 20 53 45 41 4C 3A 54 45    File: SEAL:TE
53 54 20 20 20 20 20 20 20 20 20 20 20 20 20 20 ST
20 20 20 20 20 20 20 20 20 20 31 39 38 31 2D 30           1981-0
38 2D 30 30 2F 30 30 2E 30 30 2E 30 30 20 20 20 8-00/00.00.00
```

Note that each integer has two bytes reserved per integer.

Ex. 6

Program TEST9 uses the SEEK command to move the file pointer to the fifth record. The content of the file from that point on are displayed on the console.

```
PROGRAM TEST9 (INTG);
 VAR
  TEST:FILE OF INTEGER;
  K,J:INTEGER;
 BEGIN
  J:=0;
  RESET(TEST,'SEAL:INTG');
  K:=4;
  WHILE NOTE EOF(TEST) DO
   BEGIN
    SEEK(TEST,K);
    GET(TEST);
    J:=TEST^;
    WRITELN(J);
    K:=K+1
   END  (* WHILE *)
END. (* TEST9 *)
```

1

# SECTION 11

# PROCEDURES AND FUNCTIONS

SECTION 11

# PROCEDURES AND FUNCTIONS

## 11.1 GENERAL FORM

There are two types of subroutines in PASCAL: procedures and functions. Although they are designed to serve different functions, both are declared with the same format:

```
<heading>;
|declarations;|
<compound statement>;
```

This exactly replicates the format for a program (see Section 2.1). All the parts of a procedure or function are the same as those described in the preceding sections except the heading. The headings for procedures and functions are not the same so they will be discussed in Sections 11.2 and 11.3 respectively. The END statement for both is followed by a semicolon (;) instead of the period that follows a program's END statement. The statement block is a compound statement which can contain any sequence of simple or compound statements.

When a procedure or function is called, the variables in the declarations area are placed on the stack. These local variables parameters (if any), and globally allocated variables are accessible from the procedure or function. Upon exit, the space on the stack that was allocated by the call is returned.

Since the stack is used for local variables and parameters, each new call generates a new set of variables. Therefore, procedures and functions can be called recursively. This means that each call generates a new set of variables and that any subsequent call cannot access the values of a previous call unless the values are global or passed again as parameters.

## Subprogram Placement

All identifiers in a program must be defined before they can be used. The same is true for functions and procedures. Therefore a subprogram can only call procedures or functions that precede it in the program. Those that follow it would be considered undefined and would result in a compile-time error if they were accessed. This means that the order in which subroutines are defined is important.

## FORWARD Directive

There are situations when a procedure or function must be accessed before it is defined, such as when two subroutines call each other. This can be done using the directive FORWARD. The word FORWARD is not a reserved word but it has a special meaning in a heading. Elsewhere it can be used as a user-defined identifier.

The format requires that the heading of the undefined subroutine be followed directly with the directive. This is followed by the procedure(s) and function(s) that invoke it. Later, the subroutine is completely declared but the parameter list in the heading is not repeated.

A skeleton example is:

```
PROCEDURE  front (a,b : INTEGER); FORWARD;

PROCEDURE  back (c,d : REAL);
 BEGIN
   a := 1;
   b := 2;
   front (a,b)  (* calling the procedure *)
  END; (* back *)

PROCEDURE  front;
BEGIN
  back (1,2);
END; (* front *)
```

There are many standard functions and procedures available in PASCAL. They are summarized in Appendix A. They are considered to be predeclared and operate in the same manner as user-defined subroutines do.

## 11.2 PROCEDURES

Procedures affect programming situations and do not return an explicit value. Functions, on the other hand, return a value. Procedures are usually used for input and output or for manipulating data structures.

A procedure must have a heading in the form:

PROCEDURE <procedure ident>|(parameter list)|;

Each element of the parameter list has the following format:

|VAR| <identifiers> : <type>;

The procedure identifier can be any user-defined identifier. The parameter list is a series of formal parameters and their types separated from each other by semicolons. If more than one parameter is of the same type then they can be defined together (i.e. ident1, ident2, ident3 : REAL;). The parameter list and type format may be repeated as many times as necessary to include all the parameters. Only the last type is not followed by a semicolon (;).

The procedure is called by:

<procedure ident>|(expression,...)|

A list of expressions may be variables, values, or calculations that return a value. The value must be the same type as the parameter type declared in the procedure. If the parameter is a VAR parameter (see Section 11.4), the expression must be a variable. There must be as many values as there are formal parameters and the parameters should be in the same order as in the declaration. If there are no formal parameters then the parentheses in the call and the heading are omitted.

The formal parameters are used throughout the procedure as though they are known values or variables with assigned values.

Example:

```
PROGRAM  getword;
   VAR   word,buffer : STRING;    (* the limit on buffer's size is 80 *)
         number,index : INTEGER;
         endofbuffer : BOOLEAN;
         character : CHAR;

  PROCEDURE   writeword (word : STRING);
                         number : INTEGER);
    BEGIN
      WRITELN (' The ', number,' word is ', word)
    END; (* procedure writeword *)
  BEGIN
    READLN(buffer);          (* initialization *)
    index := 0;
    number := 0;
    endofbuffer := FALSE;
    WHILE (NOT endofbuffer) DO
      BEGIN
        first := index;     (* keep the position of the first letter *)
        character := buffer[index];
        WHILE ((index <= 80) AND (character <> ' ') DO
          BEGIN
            index := index + 1;  (* check characters in buffer until *)
            character := buffer[index];  (* a blank is found or the *)
          END; (* WHILE *)       (* end of buffer found *)
        word := COPY(buffer, first, index-first-1)
        number := number + 1;    (* count number of words *)
        writeword(word,number);
        IF (index >= 80) THEN endofbuffer = TRUE
      END (* WHILE *)
  END. (* getword *)
```

## 11.3 FUNCTIONS

A function always has an associated type and returns a value of this type. It is usually used to calculate a value that must be found in several different places.

The format for a function is:

```
FUNCTION <function ident>|(parameter list)|:<type>;
```

The function identifier can be any user-defined identifier. The parameter list is a series of formal parameters in the same format as a procedure's parameter list. Only the last type in the parentheses is not followed by a semicolon. The value that is returned is sent through the function identifier and is of the type given outside of the parentheses. For this reason, the function identifier must be assigned a value somewhere in the function. Its type must be scalar, subrange or pointer.

The function is called by:

```
<function ident>(list of expressions)
```

The call may be placed anywhere a value of the function's type may legally appear. There must be as many expressions or variables as there are formal parameters. Also, if the parameter is a VAR parameter, there must be a corresponding variable in the call. If there are no parameters in the heading, the parentheses are omitted in both the heading and the call.

Example:

```
PROGRAM power;
VAR     mass,acceleration : REAL;

FUNCTION force(m,a : REAL) : REAL;
  BEGIN
    force := m * a
  END; (* mtimesa *)
```

```
  BEGIN
    READ(mass,acceleration);
    WRITELN (force(mass,acceleration))
  END. (* power *)
```

## 11.4 GLOBAL AND LOCAL VARIABLES

A variable is defined in the routine in which its definition appears and in any procedures or functions defined within the routine. For example:

```
PROGRAM  level1;
  VAR  a,b : INTEGER;

  PROCEDURE  level2A;
    VAR   c,d : CHAR;

    FUNCTION  level3;
      VAR   e,f : BOOLEAN;
      BEGIN
      END;  (* level3 *)

    BEGIN
    END;  (* level2A *)

  FUNCTION  level2B;
    VAR  g,h : REAL
    BEGIN
    END;   (* level2B *)

 BEGIN
 END.  (* level1 *)
```

This example is only a skeleton of a program since all parameters and statement blocks are missing. However, it will serve to illustrate the point. The variables "a" and "b" are called global variables and can be accessed throughout the program and its subroutines. "c" and "d" can be accessed in the procedure "level2A" and in FUNCTION level3. "e" and "f" are local to "level3", as "g" and "h" are to "level2B". If PROCEDURE level2A called by FUNCTION level2B, "g" and "h" could not be accessed within the procedure level2A even if they were passed as parameters. "g" and "h" will be undefined when level2B is initially called and they must be initiated by level2B.

If a procedure or function is passed a variable as the value for one of its parameters, any changes to that value in the subroutine would not affect the variable's value in the calling program. For example:

```
PROGRAM  outer;

  PROCEDURE  called(value : INTEGER);
   BEGIN
     value := 10
   END;  (* called *)

  PROCEDURE  test;
    VAR   value : INTEGER;
    BEGIN
      value := 5;
      called(value);    (* call procedure *)
      WRITE(value);     (* write resultant value *)
    END;  (* test *)

  BEGIN
      test
  END.  (* outer *)
```

The number that would be written out when the program executes would be "5", not "10", because the changes "called" effected would be lost after control left the procedure; however, if "value" in "called" were a VAR parameter, the result would be "10".

Varying Parameters

There are times when changes that occur in a subprogram should affect the corresponding values in the calling program. Therefore, it is possible to declare a parameter varying, meaning that any changes to it in the subprogram will be reflected in the calling routine. This is done by preceding the variable name in the called routine's parameter list with the reserved word VAR. The call to the routine is not changed except that the corresponding value must be a variable. Going back to the same example, if the heading:

```
PROCEDURE called(VAR value : INTEGER);
```

were used, the resultant value at the end of the program would be a "10". A sample program is shown on the next page.

Example:

```
PROGRAM bankstatement(Input,Output);
 VAR    id,number : INTEGER;
        balance : REAL;

 FUNCTION newbalance(trans,bal : REAL):REAL;
  BEGIN
   newbalance := bal + trans (* calculate new balance *)
  END; (* newbalance *)

PROCEDURE statement(identification,numbertrans : INTEGER;
                    VAR   balance : REAL);
 VAR      counter : INTEGER;
          transaction : REAL;
          transtype : CHAR;
 BEGIN
  FOR counter := 1 TO numbertrans DO
   BEGIN
    WRITELN('Amount and type of transaction:');
    READ(transaction,transtype);
    IF (transtype='d') OR (transtype='D') (* write amount *)
     THEN WRITE(transaction:15:7,'
     ELSE BEGIN
      WRITE(transaction:30:7);
      transaction := -transaction
      END: (* ELSE *)
    balance := newbalance(transaction,balance); (* call function *)
    WRITELN(balance:20:7)  (* write balance *)
  END  (* FOR *)
 END;  (* statement*)
 BEGIN
  REPEAT
   WRITELN;
   WRITELN('ID number, number, number of transactions, and balance');
   READLN(id,number,balance);
   statement(id,number,balance);    (* call procedure *)
   IF balance < 0.0 THEN WRITELN('Account overderawn by $',balance)
  UNTIL id = 0
 END.  (* bankstatement *)
```

# SECTION 12
# PASCAL INTRINSICS

## SECTION 12
## PASCAL INTRINSICS

### 12.1 INTRODUCTION

Intrinsics in the context of Monroe PASCAL are built-in functions always available with the system which perform specific mathematical, string, input/output, character array manipulations or miscellaneous operations. A user program can include a call to an intrinsic whenever it requires the execution of any of these operations. These functions can save a great deal of coding time. They enable the user to include the function without having to know the details behind them.

This section discusses five types of intrinsics:

1. String
2. Input/output
3. Character array manipulation
4. Mathematical
5. Miscellaneous

### 12.2 STRING INTRINSICS

PASCAL contains predefined functions and procedures that are designed to manipulate strings. Table 12-1 summarizes the string intrinsics that are available.

Table 12-1. String Intrinsics

| Heading | Description |
|---|---|
| CONCAT | Concatenates one or more strings together. |
| COPY | Returns a string copied from another string. |
| DELETE | Removes characters from a string. |
| INSERT | Inserts one string into another. |
| LENGTH | Returns the length of a given string. |
| POS | Returns the position of the first occurrence of a character sequence within a string. |

## CONCAT Function

Function: Returns a string which in the concatenation of all the strings passed to it.

Definition: FUNCTION CONCAT(<string1>|,string2,...|:STRING):STRING

Calling Format: CONCAT (<string1>|,string2,...|)

Arguments: All the arguments may be predefined string variables or strings of characters enclosed in single quotes. There may be two or more strings.

Use: This is used to join several strings into one long string.

Note: There must be at least two strings. All strings are separated by commas. The concatenated string must be smaller than 212 characters or a run-time error will occur.

Examples:

```
charstring := 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';
numstring := '1234567890';
otherstring := '#$,;*.:!?/';
```

Ex. 1

```
WRITELN('The alphanumeric characters are ',CONCAT
        (charstring,numstring));
output:  The alphanumeric characters are ABCDEFGHI
         JKLMNOPQRSTUVWXYZ1234567890
```

Ex. 2

```
str := CONCAT(numstring,otherstring);
WRITELN(str);
output:  1234567890#$,;*.:!?/
```

## COPY Function

**Function:** Returns a string copied from a specified string which contains all or part of that string.

**Definition:** FUNCTION COPY(<string>:STRING:<index>, <size>:INTEGER):STRING

**Calling Format:** COPY(<string>,<index>,<size>)

**Arguments:** The string may be any defined string variable or sequence of characters enclosed in single quotes. The index is the position of the first character to be copied. The size is the number of characters to be copied.

**Use:** COPY is often used to copy portions of a string into another one without using an index and manual incrementing.

**Note:** The index plus the size arguments must be less than or equal to the length of the string.

**Examples:**

**Ex. 1**

```
person := 'Susan Smith';
firstname := COPY (person,1,5);
WRITELN(firstname);
output:  Susan
```

**Ex. 2**

```
address := '124 Drummond Avenue, Madisonville, NJ
            07953';
zip := COPY(address,LENGTH(address)-4,5);
```

## DELETE Procedure

**Function:** Delete characters from a string.

**Definition:** PROCEDURE DELETE(<string>:STRING;<index>,<size>:INTEGER)

**Calling Format:** DELETE(<string>,<index>,<size>)

**Arguments:** The string can be any defined string variable. Thr index is the position of the first of value characters to be deleted.

**Use:** The procedure is used to save the programmer from deleting portions of a string.

**Note:** The string variable's length value is changed when the characters are removed in the DELETE procedure.

The index plus the size should not be longer than the length of the string.

**Example:**

```
overstuffed := 'This string is too long';
DELETE(overstuffed, POS('to',overstuffed),4);
WRITELN(overstuffed);
output:  This line is long.
```

## INSERT Procedure

**Function:** Inserts characters into a string.

**Definition:** PROCEDURE INSERT(<source>:STRING;VAR<destination> :STRING;<index>:INTEGER)

**Calling Format:** INSERT(<source string>,<destination string>,<index>)

**Arguments:** The source string can be either a defined string variable or a series of characters enclosed in quotes. The index is an integer number that represents the position where the source string will be inserted in the destination string.

**Use:** This is the easiest way to insert characters into a string.

**Note:** A compile time error will occur if the destination string is not a variable.

The character in the indexth position will appear after the insertion.

**Example:**

```
word := 'Fascinated.';
INSERT('ion unlimit',word,9);
WRITELN(word);
output:  Fascination unlimited.
```

## LENGTH Function

**Function:** Returns the number of characters in a string.

**Definition:** FUNCTION LENGTH(<string>:STRING):INTEGER

**Calling Format:** LENGTH(<string>)

**Argument:** The string may be a declared string variable or a series of characters enclosed by single quotes.

**Use:** It is used to determine a string's length. It is especially useful in working with buffers.

**Examples:**

**Ex. 1**

```
numberstring := '132479';
WRITELN(LENGTH(numberstring):10);
output:  6
```

**Ex. 2**

```
IF(LENGTH(buffer) = maxbufferlength) THEN
  WRITELN('Buffer overflow.');
```

## POS Function

Function: Returns the position of the first character in the first occurrence of a pattern in a string.

Definition: FUNCTION POS(<pattern>,<string>:STRING):INTEGER

Calling Format: POS(<pattern>,<string>)

Arguments: The pattern is a character or string enclosed in quotes. <string> is the text that is being scanned.

Use: The position of a character in a string is required for INSERT and DELETE.

Note: If the pattern is not found, a zero will be returned. The position of the first character is one.

Examples:

Ex. 1

```
WRITELN(POS('tu','congratulations'));
output:  7
```

Ex. 2

```
str := 'Keep something hidden.';
DELETE(str,POS('s',str),
POS('g',str)-POS('s',str) +2);
WRITELN(str);
output:  Keep hidden.
```

## 12.3 INPUT AND OUTPUT INTRINSICS

Almost every program performs some I/O, often involving files. The functions and procedures discussed in this section make it possible to do so. Table 12-2 summarizes these intrinsics.

Table 12-2. Input/Output Intrinsics

| Heading | Description |
|---|---|
| BLOCKREAD | Transfers blocks of data from a file to an array and returns the byte count. |
| BLOCKWRITE | Transfers blocks of data from an array to a file and returns the byte count. |
| CLOSE | Closes and deletes files. |
| EOF | Returns a True value when the end of a file is reached. |
| EOLN | Returns a True value when the end of a line is reached. |
| GET | Reads data from a file. |
| INP | Reads a value from a port. |
| IORESULT | Holds the result codes of I/O operations. |
| OUT | Writes a value to a port. |
| PAGE | Sends a page carriage control to a text file. |
| PUT | Writes data to a file. |
| READ | Reads data from a file or the keyboard, but does not search for an end-of-line. |
| READLN | Reads a line of input until the first position of the next line. |
| RESET | Prepares a file to be read. |
| REWRITE | Prepares a file to be written. |
| SEEK | Changes the order in which data is accessed from a file. |
| WRITE | Outputs variables and strings but does not send the cursor to a new line when it has completed. |
| WRITELN | Outputs a line and a carriage return. |

## BLOCKREAD Function

**Function:** Transfers data from a file into an array and return the count on the number of bytes that were actually read.

**Definition:** FUNCTION BLOCKREAD(<fd>:FILE;<array ident> :ARRAY;<number of blocks>:INTEGER|,first block :INTEGER|):INTEGER

**Calling Format:** BLOCKREAD(<fd>,<array id>,<block count|,first block|)

**Arguments:** The fd is the file descriptor (see Section 1.4). It cannot be defined as TEXT, though it may be of type CHAR. The array identifier is any user-defined array with the same type as the file. Its length should be an integer multiple of the number of values per block, i.e. 128 for INTEGER and CHAR files and 64 for REAL files. The <number of blocks> is the integer number of blocks that need to be transferred. Firstblock is an integer that indicates the block relative to the start of the file that should be read first. The file always starts with block zero and is read sequentially.

**Use:** This function easily manipulates blocks of unformatted data.

**Note:** There is no automatic range checking performed on the array. If it is too large, it may be filled with garbage. If it is too small, some of the information will be lost.

**Example:**

```
I := BLOCKREAD(testfile,thenline,1,1);
WRITELN(I);
output:  256
note:  This would read the second block of
       "testfile" and put it in the array
       "thenline".
```

## BLOCKWRITE Function

**Function:** Transfers data from an array into a file and returns the count of the number of bytes that were actually transferred.

**Definition:** FUNCTION BLOCKWRITE(<fd>:FILE;<array ident>:ARRAY;<number of blocks>:INTEGER|,firstblock:INTEGER|):INTEGER

**Calling Format:** BLOCKWRITE(<fd>,<array ident>|,<block count>|, first block|)

**Arguments:** The fd is the file descriptor (see Section 1.4). It cannot be defined as TEXT. The array identifier is any user-defined array with the same type as the file. Its length should be any integer multiple of the number of values per block, i.e. 128 for INTEGER and CHAR files and 64 for REAL files. The <number of blocks> is the integer number of blocks that need to be transferred. Firstblock is an integer that indicates the block relative to the start of the file that should be written to first. The file always starts with block zero and is read sequentially.

**Use:** This function easily manipulates blocks of unformatted data.

**Note:** There is no automatic range checking performed on the array. If it is too large, not all the information will be transferred. If it is too small, garbage will be used as fill-in.

**Example:**

```
I := BLOCKWRITE(testfile,thenline,2);
WRITELN(I);
output:  512
```

Note: This would transfer "thenline" into two blocks of "testfile" starting wherever the file pointer is positioned.

## CLOSE Procedure

**Function:** Closes and deletes files.

**Definition:** PROCEDURE CLOSE(<fd>:FILE|,PURGE|)

**Calling Format:** CLOSE(<fd>|,purge|)

**Arguments:** The fd is the file descriptor (see Section 1.4).

**Use:** CLOSE closes files that have been opened in a program or deletes files so they can be rewritten.

**Note:** If the file is not open, the procedure will have no effect. A closed file cannot be deleted by CLOSE.

**Example:**

```
PROGRAM testvalues(input,output,next);
 VAR    next:FILE OF INTEGER;

 BEGIN
   REWRITE(next,'PAS:next');
   IF (IORESULT <> 0) THEN
   BEGIN
       RESET(next,'PAS:next');
       CLOSE(next,PURGE);    (* close and delete *)
       REWRITE(next,'PAS: next')
   END;
     .
     .
     .
   CLOSE(next)      (* close *)
 END:
```

## EOF Function

**Function:** Returns a boolean value indicating whether the end of specified file has been reached.

**Definition:** FUNCTION EOF(<fd>:FILE):BOOLEAN

**Calling Format:** EOF(<fd>)

**Arguments:** The fd is any user-defined file descriptor (see Section 1.4).

**Use:** EOF is used when a file is being read to avoid errors. EOF is false immediately after file is opened and true on a closed file.

**Example:**

```
WHILE NOT EOF(DATA:testfile) DO
  BEGIN...
```

## EOLN Function

**Function:** Returns a boolean value indicating whether the pointer for a specified text file in at end of a line.

**Definition:** FUNCTION EOLN(<fd>:TEXT FILE):BOOLEAN

**Calling Format:** EOLN(<fd>)

**Arguments:** The fd is the user-defined file descriptor (see Section 1.4).

**Use:** The EOLN function determines if the end of a line has been reached in a textfile. EOLN returns a false value immediately after the file is opened and true on a closed file.

**Note:** The file must be TEXT or the EOLN function will have unexpected results.

**Example:**

```
chr := testfile;
WHILE NOT EOF (testfile) DO
  BEGIN
    IF EOLN(testfile)
      THEN WRITELN
      ELSE WRITE(chr);
    GET (testfile);
    chr := testfile
  END;  (* WHILE *)
```

## GET Procedure

**Function:** Reads data from a file.

**Definition:** PROCEDURE GET (<fd>:FILE)

**Calling Format:** GET(<fd>)

**Argument:** The fd is the file descriptor (see Section 1.4).

**Use:** The GET procedure is the only routine that can retrieve a value from a file that is not of type TEXT.

**Note:** A pointer variable is associated with and implicitly defined by the file. It is of the form:

<fd> ^

A buffer is assigned to the variable name into which the value is being read. The buffer is updated automatically by GET. Note that only one buffer is allocated per file.

The procedure must be preceded by a RESET which prepares the file to be read and initializes the buffer.

**Example:**

```
RESET(out);      (* Pointer moves to 1st value and
                   reads it *)
                 (* then moves pointer to 2nd value *)
WHILE NOT EOF(out) DO
  BEGIN
    X := out^; (* assign value *)
    GET(out);  (* reads next record *)
    WRITELN(X:10:5)
  END;
```

## IORESULT Function

**Function:** Returns the I/O codes giving the results of the last I/O operations.

**Definition:** FUNCTION IORESULT:INTEGER

**Calling Format:** IORESULT

**Arguments:** None.

**Use:** This function tests for errors in I/O such as during RESET and REWRITE.

**Note:** If the operation succeeds, IORESULT returns a zero. Otherwise, it returns a positive integer.

**Example:**

```
REWRITE(test,name);
IF IORESULT <> 0
  THEN BEGIN...
```

## PAGE Procedure

**Function:** Sends a top-of-form character to a file.

**Definition:** PROCEDURE PAGE(<fd>:TEXT FILE)

**Calling Format:** PAGE(<fd>)

**Argument:** The fd is the file descriptor of a TEXT file (see Section 1.4).

**Use:** PAGE is often used in text-editing programs and to make output more readable.

**Example:**

```
VAR           paper:TEXT;
              word:STRING;
BEGIN
  READLN(word);
  WHILE (word <> '') DO
    BEGIN
      IF (word <> 'page')
        THEN WRITELN (paper,word)
        ELSE PAGE (paper);
      READLN (word)
    END;  (* WHILE *)
END.
```

## PUT Procedure

**Function:** Writes a buffer to a file.

**Definition:** PROCEDURE PUT(<fd>:FILE)

**Calling Format:** PUT(<fd>)

**Argument:** The fd is the file descriptor (see Section 1.4).

**Use:** PUT is used to write to non-TEXT files.

**Note:** A pointer variable is associated with and implicitly defined by the file. Its form is:

<fd>

This variable must be assigned a buffer from the heap. The buffer is assigned a value and then it is written to the file. The buffer's contents are undefined after a PUT.

**Example:**

```
READ(X);    (* input from the screen *)
WHILE X <> 0 DO
  BEGIN
    out↑:= X;
    PUT(out);   (* written to out *)
    READ(X)     (* input from the screen *)
  END;
```

READ Procedure

Function: Reads data from a file or the keyboard and assigns it to a variable list.

Definition: PROCEDURE READ(|fd:TEXT FILE,|<variable list>)

Calling Format: READ(<fd>|,variable list|)

Argument: The fd is the TEXT file descriptor (see Section 1.4) from which the data will be read. If it is omitted, the keyboard will be used. The variable list may contain any standard or scalar data types. They must be CHAR or STRING type if a filename is used and must all be properly disclosed in the program block.

Use: The READ procedure reads values from the keyboard or from TEXT files.

Note: The READ statement will read only as many values as there are parameters. If too few identifiers are listed, not all of the desired input will be received. If too few values are listed, the computer will wait until the remaining values are entered.

If an integer variable is assigned a REAL value, the decimal part will be truncated. If a REAL variable is assigned an INTEGER value then the number will be converted to a REAL value. Both variable types assign zeroes if a character is read where a number is expected.

Example:

```
VAR       charval: CHAR;
          realnum: REAL;
          integernum: INTEGER;
BEGIN
READ (realnumber, integernum, charval);
WRITE (realnumber, ' ', integernum, ' ' charval);
   :
```

Ex. 1

```
input:  13.5   98    Letter
output:  1.3E+01 98 L
```

Ex. 2

```
input:  33     87.7  Character
output:  3.3E+01 87 C
```

Ex. 3

```
input:  champ  letter   7.5
output:  0.0E+00 0 7
```

Ex. 4

```
READ(LETTER, charval); (* LETTER is a TEXT file *)
```

## READLN Procedure

Function: Reads a line of input.

Definition: PROCEDURE READLN|(fd:TEXTFILE)|
or
PROCEDURE READLN(|fd:TEXTFILE,|<variable list>)

Calling Format: READLN|(fd)|
or
READLN(<fd>|,variable list|)

Arguments: The variable list identifier(s) may contain CHAR, INTEGER, REAL OR STRING types. Integers and reals must be terminated/separated by a space or ¶ on input.

Use: READLN is used for reading in strings. It is especially useful for text manipulations. The first format will skip the rest of a line.

Note: Caution must be used to insure that a REAL or INTEGER variable does not appear among the identifiers, or the system will either crash or assign garbage to the variable.

Examples:

```
VAR   data:STRING;
READLN(data);
WRITELN(data);
   :
```

Ex. 1

input: 'The dog came home.'
output: 'The dog came home.'

Ex. 2

input: Homeward.
output: Homeward.

Ex. 3

READLN(LETTER,data); (* LETTER is a TEXT file *)

## RESET Procedure

**Function:** Prepares a file to be read.

**Definition:** PROCEDURE RESET(<fd>:FILE|,title:STRING|)

**Calling Format:** RESET(<fd>|,title|)

**Arguments:** The fd is the file descriptor (see Section 1.4). The title is a string or string variable of the form:

'<fd>'

The two <fd's> should be the same.

**Use:** The file pointer is set to the first element in the file and prepares it to be read from. If the title is included, the file is opened before the pointer is changed.

**Note:** RESET will only open a file if the title portion is included. Also, if the file does not already exist, RESET will not create or open it.

If the file is open and another RESET is attempted, an error will be returned in IORESULT and the file status will remain unchanged.

**Examples:**

Ex. 1

```
name := 'DATA:testfile';
RESET(testfile,name);
```

Ex. 2

```
RESET(testfile);
READ(testfile,value);
```

## REWRITE Procedure

Function: Create files and prepare them for writing.

Definition: PROCEDURE REWRITE(<fd>:FILE;<title>:STRING)

Calling Format: REWRITE(<fd>,<title>)

Arguments: The fd is the file descriptor (see Section 1.4). The title is a string variable of the form:

'fd'

The two <fd's> should be the same.

Use: The file is created, its pointer is set to the first position in the file, and it is prepared for writing.

Note: REWRITE will return an error in IORESULT if the file already exists and the file will not be opened. For this reason, REWRITE can only be used once in a program in reference to a file unless it is deleted in the program.

Example:

```
name := 'DATA:test';
REWRITE(test,name);
IF (IORESULT <> 0)
  THEN BEGIN
    RESET(test,name);
    CLOSE(test,PURGE);
    REWRITE(test,name)
  END;
```

## SEEK Procedure

**Function:** Changes the order that data is accessed from a file.

**Definition:** PROCEDURE SEEK(<fd>:FILE;<record number>: INTEGER)

**Calling Format:** SEEK(<fd>,<record number>)

**Arguments:** The fd is the file descriptor (see Section 1.4). The record number is the number of the record being sought, relative to the start of the file. It must be a positive integer. The first record number is zero.

**Use:** It is used either to read or write from a place that is not the start of the file.

**Note:** The file can be of any type except TEXT. A record is defined as the structure (either simple or complex) of the file type. For example, a single value is a record in a file of REAL's, an array is a record in a file with an array as its base type.

**Example:**

**Ex. 1**

```
VAR     files : FILE OF REAL;
        temp : REAL;
   :
SEEK (files,1);
(* points to the second real value in "files" *)
   :
```

Ex. 2

```
TYPE      recks = RECORD
                    link : INTEGER;
                    data : REAL;
                  END;
VAR       filename : FILE OF recks;
          tempreck : recks;
          con : INTEGER;
BEGIN
  (* put data in the file *)
  con := 2;
  SEEK(filename, con);
  GET(filename);
  tempreck := filename  ;
  WRITELN (tempreck.link, tempreck.data);
END.

input:  1   1.5
        2   3.1
        4   5.7
        0  -9.4
output:  4 5.7
```

WRITE Procedure

Function: Outputs variables and strings.

Definition: PROCEDURE WRITE(|fd:TEXT FILE,|<item list>

Calling Format: WRITE(<fd>|,item list|)

Arguments: The fd is the file descriptor (see Section 1.4). The <item list> may be any INTEGER, REAL, or CHAR identifier or a character string inclosed in quotes. The items in the item list can be represented as:

<item 1|,item 2,...|>

An item can either be a -
<string expression> (a string variable or a character string inclosed in quotes).
or
<expression>|:field width|:precision||
which is used to format numeric output.

Field width is an integer constant that specifies the number of character positions to use in displaying the value. The default is the minimum number needed to express the value.

Precision is an integer or constant from one to six charcters that specifies the number of decimal places to be used. The default is one integer.

Use: The WRITE statement is used to output a program's results. It is also used to document output through character strings.

Note: WRITE starts writing wherever the cursor is. To start on a new line, refer to the WRITELN statement. It does not leave spaces between outputted values.

Examples:

```
   :
amount:= 55.347*10.0;
counter:= 455;
name:= 'Homer';
```

Ex. 1

```
WRITE(amount,' ' ,counter,' ', name);
output:  5.5E+02 455 Homer
```

Ex. 2

```
WRITE(amount:15:6, amount:15:4);
output:  5.53470E+02    5.534E+02
```

Ex. 3

```
WRITE(counter:15, amount:15:3);
output:             455     5.53E+02
```

Ex. 4

```
WRITE(counter,' ', name);
output:  455 Homer
```

Ex. 5

```
WRITE(LETTER,name); (* LETTER is a TEXT file *)
```

Ex. 6

```
RESET(PN,'PR:');
WRITE(PN,amount,' ', counter,' ',name);
WRITELN(PN);
5.5E+01 455 HOMER (Result on printer)
```

## WRITELN Statement

**Function:** Outputs a line and a carriage return.

**Definition:**

PROCEDURE WRITELN|(fd:TEXT FILE)|
  or
PROCEDURE WRITELN(|fd:TEXT FILE,|<item list>

**Calling Format:**

WRITELN|(<fd>)|
  or
WRITELN(<fd>|,item list|)

**Arguments:** The fd is the file descriptor (see Section 1.4). The <item list> may be any INTEGER, REAL, or CHAR identifier or a character string inclosed in greater. The items in the item list can be represented as:

<item 1|, item 2,...|>

An item can either be a -
<string expression> (a string variable or a character string inclosed in greater)
  or
<expression>|:fileid width|:precision||
which is used to format numeric output.

Field width is an integer constant that specifies the number of character positions to use in displaying value. The default is the minimum number needed to express the value.

Precision is an integer or constant from one to six characters that specifies the number of decimal places to be used. The default is one digit.

Use: The first form of the statement skips to the next line. It is used to skip a line or to begin output on a new line.

The second form is used to output a series of values and character strings. It is often used as a prompt in interactive programs.

Note: WRITELN stops on the first space of the second line. If a second line is being outputted, it will start in the second print position.

Examples:

```
  :
amount:= 55.347;
counter:=455;
name:='Homer';
  :
```

Ex. 1

```
WRITE(amount);
WRITELN(counter);
WRITELN(name);

output:  5.5E+01455
         Homer
```

Ex. 2

```
WRITE(name);
WRITELN;
WRITELN(amount,counter);

output:  Homer
         5.5E+01455
```

Ex. 3

```
WRITELN:
WRITELN ('His name is ',name,'.');

output:
         His name is Homer.
```

Ex. 4

```
WRITELN(LETTER,name); (* LETTER is a TEXT file *)
```

Ex. 5

```
RESET(PN,'PR:');
WRITELN(PN, amount,' ',counter,' ',name);
5.5E+01 455 Homer (Result on printer)
```

## 12.4 CHARACTER ARRAY MANIPULATION INTRINSICS

Character arrays are often difficult to manipulate, especially when they are packed. The intrinsics in this section simplify array manipulations. However, they require a thorough understanding of arrays in PASCAL.

These intrinsics are all byte oriented. Use them with care as no range checking is performed on the kpassed parameters.

The following table summarizes the procedures presented in this section.

Table 12-3. Character Array Manipulation Intrinsics

| Heading | Description |
|---|---|
| FILLCHAR | Places a character into an array a specified number of times. |
| MOVELEFT | Moves characters from the left end of one string to the left end of another. |
| MOVERIGHT | As MOVELEFT but in the opposite direction. |
| SCAN | Finds the distance a character is from a starting point. |

## FILLCHAR Procedure

**Function:** Places a character into a packed array a specified number of times.

**Definition:** PROCEDURE FILLCHAR(<array>:ARRAY;<length>:INTEGER; <character>:CHAR)

**Calling Format:** FILLCHAR(<array>,<length>,<character>)

**Arguments:** The array must be a PACKED ARRAY of CHAR. The character is a single character enclosed in quotes or a variable of type CHAR. The length is the number of characters to place in the array. It must be an integer.

**Use:** The procedure transfers a character with only one memory reference.

**Note:** The array may be subscripted. If it is, the character will be placed in the array starting at the indexed position.

**Example:**

```
PROGRAM FILL(OUTPUT);
TYPE
   ARR=PACKED ARRAY[1..50] OF CHAR;
VAR
   ARR1    :ARR;
   I       :INTEGER;
BEGIN(*FILL*)
   FOR I:=1 TO 10 DO
       BEGIN
           ARR1[I]:='A';
           WRITELN(ARR1[I]) (* PRINTS 10 A's *)
       END;
   WRITELN;
   FILLCHAR(ARR1,5,'B');      (* REPLACES FIRST FIVE
                               A's WITH B's *)
   FOR I:=1 TO 10 DO
       WRITELN(ARR1[I])       (* PRINTS 5 B's and 5
                               A's *)
END. (* FILL *)
```

## MOVELEFT Procedure

Function: Moves a specified number of characters from the left end of one string to the left end of another.

Definition: PROCEDURE MOVELEFT(VAR<source>,<destination>:CHAR; <length>:INTEGER)

Calling Format: MOVELEFT(<source>,<destination>,<length>)

Arguments: <source> is in the source string and <destination> is in the destination string. The length is the number of characters to be moved. It must be a positive integer.

Use: The procedure is used to transfer characters from one part of a packed character array to another.

Note: <source> and <destination> may be the same array. If they are subscripted then the indexed positions are assumed to be the left ends of the strings.

Example: using:

```
VAR     str: STRING[31];
        next : STRING[11];
str := 'This is the text in this string';
next := 'Programming'; (* ALL EXAMPLES ARE BUILDING *)
```

Ex. 1

```
MOVELEFT(str[1],str[3],10); WRITELN(str);
output:  'ThThThThThThtext in this string'
```

Ex. 2

```
MOVELEFT(str[17],str[3],9); WRITELN(str);
output:  'Th in this htext in this string'
```

Ex. 3

```
MOVELEFT(str[11],str[12],1); WRITELN(str);
output:  'Th in this   text in this string'
MOVELEFT(next,str,11); WRITELN(str);
output:  'Programming text in this string'
```

## MOVERIGHT Procedure

**Function:** Moves a specified number of characters from the right end of one string to the right end of another.

**Definition:** PROCEDURE MOVERIGHT(VAR<source>,<destination>:CHAR; <length>:INTEGER)

**Calling Format:** MOVERIGHT(<source>,<destination>,<length>)

**Arguments:** <source> is in the source string and <destination> is in the destination string. The length is the number of characters to be moved. It must be a positive integer.

**Use:** The procedure is used to transfer characters from one part of a packed character array to another.

**Note:** <source> and <destination> may be in the same array. If they are subscripted then the indexed positions are assumed to be the left end of the strings.

**Examples:** using:

```
VAR     str: STRING[31];
        next : STRING[11];
str := 'This is the text in this string';
next := 'Programming'; (* ALL EXAMPLES ARE BUILDING *)
```

**Ex. 1**

```
MOVERIGHT(str[1], str[3],10) ' WRITELN(str);
output:  ThThis is thtext in this string
```

**Ex. 2**

```
MOVERIGHT(str[17],str[3],9); WRITELN(str);
output:  Th in this htext in this string
```

Ex. 3

```
MOVERIGHT(next,str,11); WRITELN(str);
output:  Programminghtext in this string
```

Ex. 4

```
MOVERIGHT(next[1],next[5],5); WRITELN(next)
output: ProgrProgrng
```

## SCAN Function

Function: Returns the distance a character is from a specified starting point in a string.

Definition: FUNCTION SCAN(<length>:INTEGER;<partial expression>;<array>:CHAR):INTEGER

Calling Format: SCAN(<length>,<partial expression>,<array>)

Arguments: The length is a positive or negative integer. The partial expression is either an equal (=) or not equal (<>) sign followed by a character expression. The array should be a PACKED ARRAY of CHAR and may be subscripted to denote the starting point.

Use: SCAN determines the number of characters from the starting position to a character expression. It is often used in conjunction with MOVELEFT and MOVERIGHT.

Note: The value returned by the function will be either the specified length or the number of characters from the starting position to the first occurrence of the character expression. The length will be returned if the character is not in the array. If it is in the starting position the resultant value will be zero.

If the length is a negative integer, the function will scan backward from the starting position and the returned value will be negative.

Examples: Using the packed array, arr, with the value:
'There he goes again.'

Ex. 1 SCAN(15, ='T',arr) = 0

Ex. 2 SCAN(10, <> 'T',arr) =1

Ex. 3 SCAN(10, 'g',arr[6]) = 4

Ex. 4 SCAN(100, = 'z',arr) = 100

Ex. 5 SCAN(-10, = 'e',arr[10]) = -2

## 12.5 MATHEMATICAL FUNCTIONS

Monroe PASCAL contains predefined functions that perform mathematical functions. Table 12-4 summarizes the functions that are available. A detailed description of each function follows this table.

Table 12-4. PASCAL Mathematical Functions

| Function | Description |
|---|---|
| ABS | Returns the absolute value of a value. |
| ARCTAN | Returns the arctangent of a value. |
| COS | Returns the COS of a value. |
| EXP | Returns the exponential function of a value (i.e., $e^{value}$). |
| LN | Returns the material logarithm of a value (i.e., log $e^{value}$). |
| LOG | Returns the common logarithm (base 10) of a value. |
| MOD | Returns the remainder when one integer is divided by another. |
| ODD | Returns a BOOLEAN value specifying whether an integer is odd. |
| ROUND | Returns the integer representation of a REAL number (rounded). |
| SIN | Returns the sine of a value. |
| SQR | Returns the square of a number. |
| SQRT | Returns the square root of a number. |
| TRUNC | Returns the INTEGER representation of the decimal portion of a REAL number (truncated). |

## ABS Function

Function: Returns the absolute value of a number.

Definition: FUNCTION ABS(<value>:REAL or INTEGER):REAL or INTEGER

Calling Format: ABS(<value>)

Argument: The value may be any constant, variable, or expression that represents a number.

Use: The ABS function is used when the value being sought must be positive. It is often used in conjunction with SQRT.

Note: The type of the output will be the same as the input type.

Examples:

Ex. 1

```
WRITELN(ABS(-1):10,ABS(-2.5):10,ABS(5.2):10);
output:    1     2.5     5.2
```

Ex. 2

```
VAR    length,X1,X2,Y1,Y2:REAL;
         :
length := SQRT(ABS(SQR(X2-X1)+SQR(Y2-Y1)));
```

## ARCTAN Function

Function: Returns the value of the arctangent of a number.

Definition: FUNCTION ARCTAN(<value>:REAL):REAL

Calling Format: ARCTAN(<value>)

Argument: The value may be a numeric constant, a number, a variable with a numeric value or an expression.

Use: The ARCTAN function is used for trigonometric calculations.

Note: The ARCTAN function acts on a radian value and returns a REAL value in radians.

Examples:

Ex. 1

```
WRITELN(ARCTAN(0.5):13:4);
output:  4.636E-0
```

Ex. 2

```
VAR arctanY,Y:REAL;
  :
arctanY := ARCTAN(Y);
```

## COS Function

**Function:** Returns the cosine of a value.

**Definition:** FUNCTION COS(<value>:REAL):REAL

**Calling Format:** COS(<value>)

**Argument:** The value may be a numeric constant, a number, a variable with a numeric value or an expression.

**Use:** The COS function is used for trigonometric calculations.

**Note:** The value is in radians, not in degrees. The value that is returned is REAL and should be formatted for greater accuracy.

**Examples:**

**Ex. 1**

```
WRITELN(COS(0.5):13:4);
output:  8.776E-01
```

**Ex. 2**

```
VAR       X,tanX:REAL;
      .
      .
      .
X := 3;
tanx := SIN(x)/COS(x);
```

## EXP Function

Function: Returns the exponential function of a value.

Definition: FUNCTION EXP(<value>:REAL):REAL

Calling Format: EXP(<value>)

Argument: The value may be a number, a numeric constant, a variable with a numeric value or an expression.

Use: The EXP function is used in calculations that involve the factor e.

Note: The most common mathematical representation of EXP(X) is $e^x$.

Examples:

Ex. 1

```
WRITELN(EXP(.5):13:4);
output:  1.649E00
```

Ex. 2

```
VAR E;X:REAL;
   .
   .
   .
E := 5.0*EXP(X)+2*x;
```

## LN Function

**Function:** Returns the natural logarithm of a value.

**Definition:** FUNCTION LN(<value>:REAL):REAL

**Calling Format:** LN(<value>)

**Argument:** The value must be a number greater than zero.

**Use:** The LN function is often used in calculations involved in graphing.

**Examples:**

**Ex. 1**

```
WRITELN(LN(0.5):13:4);
output:  -6.931E-01
```

**Ex. 2**

```
VAR    X,Y:REAL;
  .
  .
  .
Y := LN(2*X+5.0);
```

## LOG Function

Function: Returns the logarithm of a number.

Definition: FUNCTION LOG(<value>:REAL):REAL

Calling Format: LOG(<value>)

Argument: The value may be a number, a numeric constant, a variable with a numeric value or an expression.

Use: Logarithms are often used to simplify arithmetic on very large or very small numbers.

Note: The function returns a REAL value.

Examples:

Ex. 1

```
WRITELN(LOG(0.5):13:4);
output:  -3.010E-01
```

Ex. 2

```
VAR    Y,X:REAL;
  :
Y := 2*LOG(X+5);
```

## MOD Function

**Function:** Returns the remainder when two integers are divided.

**Definition:** FUNCTION(<value>:INTEGER MOD <value>:INTEGER): INTEGER

**Calling Format:** <value> MOD <value>

**Argument:** Both values may be any constant, variable, or expression that represents an INTEGER.

**Use:** Since it finds the remainder after division, MOD is often used to test if the division came out even.

**Note:** The first value is divided by the second. Therefore, the second number cannot be equal to zero. Also, anything MOD one will always equal zero.

If the first value is positive, the result is positive. If it is negative, the result is negative. This is regardless of the value of the second value.

**Examples:**

**Ex. 1**

```
WRITELN(2 MOD 3:10,3 MOD 2:10 -4 MOD 3:10, -5 MOD-2
        :10);
```

```
output:   2     1     -1     -1
```

Ex. 2

```
VAR     I:INTEGER;
        IVAL : STRING[5];
    .
    .
    .
IF(I MOD 2 = 0)
   THEN IVAL := 'EVEN'
   ELSE IVAL := 'ODD';
```

## ODD Function

**Function:** Returns a BOOLEAN value specifying when an integer is odd.

**Definition:** FUNCTION ODD(<value>:INTEGER):BOOLEAN

**Calling Format:** ODD(<value>)

**Argument:** The value may be any constant, variable, or expression that represents an INTEGER value.

**Use:** The ODD function is often used to determine if a group has an even or odd number of elements. If there are an odd number of elements this function returns a true value. This is especially useful in calculating medians and the like.

**Example:**

```
VAR    counter:INTEGER;
       truth:CHAR;
          .
          .
          .
IF  ODD(counter)
    THEN truth := 'Y'
    ELSE truth := 'N';
```

## ROUND Function

**Function:** Returns the INTEGER representation of a REAL number by rounding it to the closest integer.

**Definition:** FUNCTION ROUND(<value>:REAL):INTEGER

**Calling Format:** ROUND(<value>)

**Argument:** The value may be any constant, variable, or expression that represents a REAL number.

**Use:** ROUND is often used to increase the accuracy when converting a REAL to an INTEGER.

**Note:** If the REAL value has a five in the tenths place and the value is positive, PASCAL will round up. If it is negative, it will round down.

**Examples:**

Ex. 1

```
WRITELN(ROUND(1.3):10,ROUND(1.6):10,ROUND(1.5):10,
        ROUND(-2.5):10);
output: 1     2     2     -3
```

Ex. 2

```
VAR    X:REAL;
   .
   .
   .
IF ROUND(X) = TRUNC(X) THEN ...
```

## SIN Function

**Function:** Returns the sine of a value.

**Definition:** FUNCTION SIN(<value>:REAL):REAL

**Calling Format:** SIN(<value>)

**Argument:** The value may be a numeric constant, a number, a variable with a numeric value, or an expression.

**Use:** The SIN function is used in calculating trigonometric functions.

**Note:** The value must be in radians, not in degrees. When the value is outputted, it should be formatted for greater accuracy.

**Examples:**

**Ex. 1**

```
WRITELN(SIN(0.5):13:4)
output:  4.794E-01
```

**Ex. 2**

```
VAR      X:REAL;
     :
X := 1;
WRITELN(SIN(x):13:4);
output:  8.415E-01
```

## SQR Function

**Function:** Returns the square of a value.

**Definition:** FUNCTION SQR(<value>:REAL or INTEGER):REAL or INTEGER

**Calling Format:** SQR(<value>)

**Argument:** The value may be a number, a numeric constant, a variable with a numeric value, or an expression.

**Use:** Numbers are often squared in calculations.

**Note:** If the value that is squared is REAL, the result will be REAL; if it is INTEGER, the result will be an integer.

**Examples:**

**Ex. 1**

```
WRITELN(SQR(5.2):13:4,SQR(7):10);
output:   2.704E+01     49
```

**Ex. 2**

```
VAR    X,Y,result:  REAL;
  :
result := SQR(X-1)+SQR(Y-4);
```

## SQRT Function

**Function:** Returns the square root of a number.

**Definition:** FUNCTION SQRT(<value>:REAL):REAL

**Calling Format:** SQRT(<value>)

**Argument:** The value must be greater than or equal to zero.

**Use:** Many square roots of numbers are taken in calculations. Perhaps the best-known example is in the formula for the distance between two points in graphing.

**Examples:**

Ex. 1

```
WRITELN(SQRT(.5):13:5);
output:   7.071E-01
```

Ex. 2

```
VAR    X,Y,dist:REAL;
   :
dist := SQRT(SQR(X)+SQR(Y));
```

## TRUNC Function

**Function:** Returns the INTEGER representation of the decimal portion of a REAL number which has been truncated.

**Definition:** FUNCTION TRUNC(<value>:REAL):INTEGER

**Calling Format:** TRUNC(<value>)

**Argument:** The value may be any constant, variable, or expression that represents a REAL number.

**Use:** It is often used when converting REALs to INTEGERs for further calculations.

**Note:** The input to the function is REAL but the output is INTEGER. The result is not necessarily the integer that is closest to the input.

**Examples:**

**Ex. 1**

```
WRITELN(TRUNC-0.2);10,TRUNC(2.6):10);
output:   0          2
```

**Ex. 2**

```
VAR    in:REAL;
out:  INTEGER;
   .
   .
   .
out := TRUNC(in*2);
```

## 12.6 MISCELLANEOUS ROUTINES

The functions and procedures presented in this section are useful in diverse applications of PASCAL. They are summarized in Tables 12-5 and 12-6.

Table 12-5. Miscellaneous Intrinsics

| Item | Description |
|---|---|
| DATE | Gives the date. |
| DISPOSE | Returns allocated memory to the heap. |
| EOLNCHR | Returns an integer value representing a termination. |
| EXIT | Results in an orderly exit. |
| GOTOXY | Sends the cursor to specified positions on the screen. |
| HALT | Terminates the execution of a PASCAL program. |
| MARK | Sets a pointer to the current top-of-heap of available memory. |
| NEW | Allocates space from the heap. |
| OPTION | Returns the starting switches. |
| RELEASE | Sets the top-of-heap pointer for the available memory to the specified pointer. |
| SIZEOF | Returns the number of bytes a variable or type identifier represents. |
| STARTPAR | Holds the starting parameters. |
| SVC | Executes Supervisor Calls. |
| TIME | Gives the time since the system was last booted. |
| PWROFTEN | Returns a REAL result of the number 10 raised to the power of the integer parameter supplied. |
| OUT | Writes a value to a point. |
| INP | Returns an integer value from an I/O port. |

Table 12-6. Logical Intrinsics

| Item | Description |
|---|---|
| IXOR | Performs exclusive OR operation. |
| IOR | Performs OR operation. |
| IAND | Performs AND operation. |
| ISHIFT | Returns an integer result from a shift operation. |
| ISWAP | Returns an integer with low and high byte swapped. |

## DATE Function

**Function:** Returns the date.

**Definition:** FUNCTION DATE:STRING

**Calling Format:** DATE

**Arguments:** None.

**Use:** The DATE function can be used to set switches or to date program corrections.

**Note:** The date is returned in a string in the format "YYYY-MM-DD" where the Y's represent the year, the M's represent the month, and the D's represent the day.

The date must be set every day.

**Example:**

```
WRITELN(DATE);
output:  1981-09-04
```

## DISPOSE Procedure

Function: Returns allocated memory to the heap.

Definition: PROCEDURE DISPOSE(<ptr>:POINTER)

Calling Format: DISPOSE(<ptr>)

Arguments: <ptr> is a dynamic pointer variable previously allocated memory using NEW(<ptr>).

Use: To return a linked list of free space to the heap.

Note: When the DISPOSE procedure is executed, the memory is placed onto the linked list of available free space. This list is then searched for a suitably large space when a NEW is executed. The list is cleared when a RELEASE is executed.

Example:

```
TYPE     pointer =   into;
         into = RECORD
           link : pointer;
           data : STRING[25]
         END;
VAR   next : pointer;
BEGIN
  NEW(next);        (* allocate *)
     :
  DISPOSE(next);    (* deallocate *)
END.
```

## EOLNCHR Function

**Function:** Returns an integer which is a termination character detected by READLN in the last line of input from the console.

**Format:** EOLNCHR

**Use:** EOLNCHR is used to detect the CR character (13 decimal) as well as the value of the function keys if they are pressed during the execution of the READLN.

The console has eight function keys labelled F1/F9 through F8/F16.

A programmer can assign various functions to the function keys, e.g., cursor movenemts, write data, read data, update data or a jump to a program module.

The function keys can produce 32 different ASCII values as shown in Table 12-6.

Table 12-7. Function Key ASCII Values

| Key | Normal | Shift | CTRL | Shift+CTRL |
|---|---|---|---|---|
| F1/F9 | 128 | 136 | 144 | 152 |
| F2/F10 | 129 | 137 | 145 | 153 |
| F3/F11 | 130 | 138 | 146 | 154 |
| F4/F12 | 131 | 139 | 147 | 155 |
| F5/F13 | 132 | 140 | 148 | 156 |
| F6/F14 | 133 | 141 | 149 | 157 |
| F7/F15 | 134 | 142 | 150 | 158 |
| F8/F16 | 135 | 143 | 151 | 159 |
| RETURN | 13 | | | |
| RUN | 208 | | | |
| LOAD | 209 | | | |
| CONTINUE | 210 | | | |
| HOME | 199 | | | |
| ↑ | 197 | | | |
| ↓ | 198 | | | |

In addition, the RETURN, RUN, LOAD, CONTINUE and certain cursor keys act as terminators.

Example:

```
program exp50(input,output);    (* TEST EOLNCHR *)
   CONST
        CR='(:13:)'
        F1='(:128:)'
   var
        index:integer;
        letter:string;
 (* program will detect CR or F1 keypress *)
   begin
        for index:=1 to 10 do
          begin
                readln(letter);
             case eolnchr of
                CR:writeln('CR pressed');
                F1:writeln('function key F1
                           pressed');
             end  (*case*)
          end  (*for*)
   end. (*exp50*)
```

## EXIT Procedure

**Function:** Results in an orderly exit from a procedure, function or main program.

**Format:** EXIT(<identifier>)

**Argument:** Identifier is the name of a procedure, function, program name or the word PROGRAM.

**Use:** Following the execution of EXIT, processing continues at the final end statement in the procedure name. The procedure name need not be the procedure currently under execution. If the procedure has not been invoked when the EXIT is executed, a run time error will occur. If the procedure identifier passed to EXIT is a recursive procedure, the most recent inovation of that procedure will be exited. When an EXIT of a function contains no assignment to the function identifier, an undefined value will be returned. EXIT brings the program to an orderly halt when the program name or reserved word PROGRAM is used as the parameter for EXIT.

**Example:**

```
program exp10;                        (* test exit *)
    var
     index:integer;
    procedure testexit;
    begin
     writeln('procedure testexit');
     index:=10;
     if index=10 then exit(testexit)
      else writeln('noexit');
    end;
    begin
     testexit;
    end.
```

## GOTOXY Procedure

**Function:** Sends the cursor to specified coordinates.

**Definition:** PROCEDURE GOTOXY(<x-coordinate>,<y-coordinate>: INTEGER)

**Calling Format:** GOTOXY(<x-coordinate>,<y-coordinate>)

**Arguments:** X-coordinate and Y-coordinate are both integers such that

$$0 \le \text{X-coordinate} \le 80$$
$$0 \le \text{Y-coordinate} \le 22$$

**Use:** GOTOXY is used extensively in graphing.

**Note:** (0,0) is the top left corner of the screen. If either coordinate goes out of range the edge coordinate (0,22, or 80) will be used instead. There is no window clipping.

**Examples:**

**Ex. 1**

```
I := 1;
READLN(X[I],Y[I];
WHILE X[I] > = 0 DO
  BEGIN
    I := I+1;
    READLN(X[I],Y[I])  (* read in coordinate pairs *)
  END;  (* WHILE *)
FOR K := 1 TO I
  DO BEGIN
    GOTOXY(X[K],Y[K]);
    WRITE('X')     (* mark the position *)
  END;  (* FOR *)
```

**Ex. 2**

```
GOTOXY(-5,27)
```

would go to position (0,22).

## HALT Procedure

**Function:** Terminates the execution of a PASCAL program.

**Format:** HALT

**Use:** The HALT statement is used to terminate the execution of a PASCAL program. The statement is normally used when a total error occurs.

When running in CSS-mode (see Section 14) an internal error is generated when the HALT statement is executed. If the CSS command $TEST has been given, the CSS-processor continues to execute CSS commands. If not, the CSS is stopped. It is possible to detect the execution of a HALT introduction by using the command "$$IF ERROR". The IF command will obtain the value true if the previous PASCAL program executed a HALT instruction.

**Example:**

```
program expl;                          (* test halt *)
    const
          unit=10;
    var
          error:boolean;
          item:integer;
    begin
          item:=10;
          if item=unit then error:=true
                else error:=false;
          if error=true then halt
                else writeln('continue');
    end.
```

## MARK Procedure

**Function:** Sets a pointer to the top-of-heap of the available free memory. The address of the heap is stored in the pointer.

**Definition:** PROCEDURE MARK(<pointer var>:POINTER)

**Calling Format:** MARK(<pointer var>)

**Arguments:** The pointer variable must be declared pointer type. See Section 10 for more details.

**Use:** MARK is used in conjunction with RELEASE to return unneeded dynamically allocated memory to the system.

**Note:** MARK should be followed by RELEASE.

**Example:**

**Ex. 1**

```
WHILE rear <> front DO
BEGIN
  WRITELN(front  .food);
  heapptr := front;
  front := front  .link; (* update the queue's pointer *)
  MARK(heapptr);  (* set heapptr to the top of the stack *)
  RELEASE(heapptr)  (* release the record *)
END;  (* WHILE *)
```

Ex. 2

```
program heaptest;
    type
         pt1=^person;
         pt2=^integer;
        person=record
             name:string[10];
             ssnum:string[10];
             address:string[10];
        end;
    var
        p, r:pt1;
        heap:pt2;
        heapcount:integer;
    begin
        mark(heap);
        new(p);
        p^.name:='john smith';
        p^.ssnum:='132-46-846';
        p^.address:='1234 104st';
        writeln(p^.name,p^.ssnum,p^.address);
        release(heap);
        writeln(p^.name,p^.ssnum,p^.address);
    end.
```

## NEW Procedure

Function: Allocates space from the heap.

Definition: PROCEDURE NEW(<ptr>:POINTER)

Calling Format: NEW(<ptr>)

Arguments: <ptr> is a dynamic pointer variable.

Use: NEW allocates space from the heap for dynamic variables.

Note: The pointer points to the free space in the heap after the NEW procedure is executed. The amount of space is determined by the type that the pointer points to.

Executing a second NEW procedure does not return old space. DISPOSE or RELEASE must be used for this.

Example:

```
TYPE    pointer =   into;
        into = RECORD
          link : pointer;
          data : INTEGER
        END;
VAR     first, next, last : pointer
BEGIN
  NEW (first);
  NEW (next);
  NEW (last);
END.
```

## OPTION Function

Function: Returns the switches that represent the options that were chosen when PASSYS or PASCAL was executed.

Definition: FUNCTION OPTION:STRING

Calling Format: OPTION

Arguments: None.

Use: This function is often used when testing an option against the current condition.

Note: There are twenty-six switches, each one bit long so OPTION returns four bytes. Each switch corresponds to a letter in the alphabet and is a one if the switch is set, zero if it is not. The bytes may not form a recognizable character so it usually cannot be written out.

For the possible switches and their meanings, look at the System programs' options in Section 13.

Example: for : PASCAL,AK userprog
BIT 0 in OPTIONS[1] and Bit 2 in OPTIONS[2]
would be 1's, the rest would be zeroes.

## RELEASE Procedure

**Function:** Sets the top-of-heap pointer to the memory location of the pointer variable.

**Definition:** PROCEDURE RELEASE(<pointer var>:POINTER)

**Calling Format:** RELEASE(<pointer var>)

**Arguments:** The pointer must be declared pointer type. See Section 9 for more details.

**Use:** RELEASE is used in conjunction with MARK to return unneeded dynamically allocated memory to the system.

**Note:** RELEASE should always follow MARK. Also, all objects allocated between the MARK and RELEASE are deallocated and should not be referenced.

**Examples:**

```
MARK(free);
NEW(X);
NEW(Y);
NEW(Z);
RELEASE(free);  (* Return space from X, Y, and Z *)
```

## SIZEOF Function

**Function:** Returns the number of bytes in memory that are assigned to an identifier.

**Description:** FUNCTION SIZEOF(<identifier>):INTEGER

**Calling Format:** SIZEOF(<identifier>)

**Arguments:** The identifier is a user-defined variable or type identifier.

**Use:** SIZEOF is particularly useful for the FILLCHAR, MOVELEFT and MOVERIGHT intrinsics.

**Note:** The result is in bytes, not characters.

**Examples:**

```
TYPE     rec:RECORD
           link:INTEGER;
           data:REAL
         END;
VAR      value:INTEGER;
         next:REAL;
         name:CHAR;
```

**Ex. 1**

```
WRITELN(SIZEOF(value):10,SIZEOF(name):10,SIZEOF(next):10);
output:   2   2   4
```

**Ex. 2**

```
WRITELN(SIZEOF(rec));
output:  6
```

## STARTPAR Function

Function: Returns the characters that are written after the code-file name when a program is executed.

Definition: FUNCTION STARTPAR:STRING

Calling Format: STARTPAR

Arguments: None.

Use: It allows the user to access the parameters that were appended to the filename.

Note: The function returns the characters as a STRING.

Examples:

```
for PASCAL PASC:userprog,ABC123
  WRITELN(STARTPAR);
output:  ABC123
```

## SVC Function

BORDE FÖRKLARAS BÄTTRE MED MERA EX.

**Function:** Executes Supervisor Calls and returns a false value if the SVC was in error.

**Definition:** FUNCTION SVC(<n>:INTEGER;<parameter block>:PACKED RECORD):INTEGER

**Calling Format:** SVC(<n>,<parameter block>)

**Arguments:** n is the SVC-number. The list of SVC's and their associated numbers are given below. The parameter block formats may be found in the Monroe Operating System Programmer's Reference Manual.

**Use:** This function allows the execution of SVC's so the operating system can be called to perform special tasks.

**Note:** The function returns a False value if the SVC was in error. Otherwise, it returns a True value.

The Supervisor Calls and their associated numbers are listed below. Each of the calls is discussed in detail in the Monroe Operating System Programmer's Reference Manual.

| n | Function |
|---|---|
| 1 | General Purpose I/O Requests |
| 2 | Memory Handling (2.1) |
| | Log Message (2.2) |
| | Pack File Descriptor (2.3) |
| | Pack Numeric Data (2.4) |
| | Unpack Binary Number (2.5) |
| | Fetch/Set Date/Time (2.7) |
| | Scan Mnemonic Table (2.8) |
| | Open/Close Device (2.12) |

| n | Function |
|---|---|
| 3 | Timer Requests |
| 4 | Task Device |
| 5 | Loader Handling |
| 6 | Task Request |
| 7 | File Request |
| 8 | Resource Handling |

Caution: Incorrect use of the SVC's can crash the system.

**Example:**

```
TYPE     line =
           RECORD
                CASE BOOLEAN OF
                  TRUE : (I : INTEGER);
                  FALSE : (S :  STRING)
           END;  (* line *)
         byte = 0..255;  (* 1 byte *)
         SVC1B = PACKED RECORD  (* Parameter block *)
           TS,LV,RS,FC : byte;
           BAD : line;
           BSZ, BCNT, RND, RND2 : INTEGER;
         END;  (* SVC1B *)

VAR      SVC1 : SVC1B;
         RESULT : BOOLEAN;
BEGIN
  (* assign values to the various fields of SVC1 *)
  RESULT := SVC(1,SVC1);
```

(Refer to the actual parameter block in the Monroe Operating System Programmer's Reference Manual for a better understanding of SVC1.)

## TIME Function

**Function:** Returns the system time or if the system time was not set, the elapsed time since the system was last booted.

**Definition:** FUNCTION TIME:STRING

**Calling Format:** TIME

**Arguments:** None.

**Use:** The TIME function helps in detecting infinite loops. It can also be used to store the time associated with a particular data entry.

The function is returned in a string in the form HH.MM.SS where "H's" represent the hour 01-23, M's represent the minutes, and S's represent the seconds.

**Note:** The time is reset automatically to 00-00-00 each time the computer is booted. To set the system time, use the TIME Utility. Refer to the 8800 Series Utility Programs Programmer's Reference Manual for details.

**Example:**

```
:
WRITELN(TIME);
output:  01.52.39
```

## INP Function

Function: Returns the integer value from a port number.

Definition: FUNCTION INP(<PORT>):INTEGER

Calling Format: INP(<PORT>);

Arguments: PORT is an I/O number. Refer to Table K-1 for the port numbers.

Use: INP is used to input an integer value from an I/O port, such as the communications interface.

Example:

```
  :
J:=INP(164);
  :
```

## OUT Procedure

Function: Writes a value to a port.

Definition: PROCEDURE OUT(<PORT>,<DATA>):INTEGER

Calling Format: OUT(<PORT>,<DATA>);

Arguments: PORT is an I/O. Refer to Table K-1 for the port numbers.

Use: This procedure is used to pass an INTEGER value to an output port, such as the communications interface.

Example:

```
⋮
OUT(164,DATA);
⋮
```

## PWROFTEN Function

**Function:** Returns a REAL result of the number 10 raised to the power of the integer parameter supplied.

**Definition:** FUNCTION PWROFTEN(<VALUE>:INTEGER):REAL

**Calling Format:** PWROFTEN(<VALUE>);

**Arguments:** VALUE is type INTEGER.

**Use:** This function converts an integer parameter to its exponential form.

**Example:**

Declaration:

```
VAR
  result:real;
  value:integer;
```

Main Section:

```
  value:=4;
  result:=pwroften(value);
  writeln(result)
  end.
```

Output:

```
  1.0E+04
```

## Logical Intrinsics

### IAND Function

**Function:** Performs a bitwise AND operation.

**Definition:** FUNCTION IAND(<VAL1>,<VAL2>:INTEGER):INTEGER

**Calling Format:** IAND(<VAL1>,<VAL2>);

**Arguments:** VAL1 and VAL2 are type INTEGER.

**Use:** IAND is used for bitwise ANDing of INTEGER values.

**Example:**

```
PROGRAM IANDC;
VAR
  RESULT   :INTEGER;
  PN       :TEXT;
BEGIN
  RESET(PN,'PR:');
  RESULT:=IAND(1,2);  (* BIT VALUE 1=00000001, 2=00000010 *)
                      (* 1 AND 2 = 00000000 = 0 *)
  WRITELN(PN,RESULT); (* RESULT=0 *)
  RESULT:=IAND(5,14); (* BIT VALUE 5=00000101, 14=00001110 *)
                      (* 5 and 14 = 00000100 = 4 *)
  WRITELN(PN,RESULT); (* RESULT=4 *)
  RESULT:=IAND(28,27);(* BIT VALUE 28=00011100,27=00011011 *)
  WRITELN(PN,RESULT)  (* RESULT=24 *)
END.
```

## IOR Function

**Function:** Performs a bitwise OR operation.

**Definition:** FUNCTION IOR(<VAL1>,<VAL2>:INTEGER):INTEGER

**Calling Format:** IOR(<VAL1>,<VAL2>);

**Arguments:** VAL1 and VAL2 are type INTEGER.

**Use:** IOR is used for bitwise ORing of INTEGER values.

**Example:**

```
PROGRAM IORC;
VAR
   RESULT  :INTEGER;
   PN  :TEXT;
BEGIN
   RESET(PN,'PR:');
   RESULT:=IOR(1,2);      (* BIT VALUE 1=00000001, 2=00000010 *)
                          (* 1 or 2=00000011=3 *)
   WRITELN(PN,RESULT);    (* RESULT =3 *)
   RESULT:=IOR(5,14);     (* BIT VALUE 5=00000101, 14=0001110 *)
                          (* 5 OR 14=00001111=15 *)
   WRITELN(PN,RESULT);    (* RESULT = 15 *)
   RESULT:=IOR(28,27);    (* BIT VALUE 28=00011100, 27=00011011 *)
                          (* 28 OR 27=00011111=31 *)
   WRITELN(PN,RESULT)     (* RESULT = 31 *)
END.
```

## ISHIFT Function

**Function:** Returns an integer result from the operation of shifting a variable left or right.

**Definition:** FUNCTION ISHIFT(<value>,<direction>:INTEGER):INTEGER

**Calling Format:** ISHIFT(<value>,<direction>);

**Arguments:** Value is type INTEGER.

Direction steps to the left or right depending on the sign of direction. The range is

-15 < direction < 15.

Positive direction is LEFT SHIFT.
Negative direction is RIGHT SHIFT.

**Use:** ISHIFT is used to shift the bit positions in the positive or negative direction. The operation is equivalent to:
2 raised to the power of 'direction' times 'value'.

**Example:**

```
PROGRAM ISHIFTC(OUTPUT);
VAR
   C    :INTEGER;
   PN   :TEXT;
BEGIN
   RESET(PN,'PR:');
   C:=ISHIFT(5,2);      (* 2 TO THE POWER OF 2 TIMES 5 *)
   WRITELN(PN,C);       (* C=20 *)
   C:=ISHIFT(3,4);      (* 2 TO THE POWER OF 4 TIMES 3 *)
   WRITELN(PN,C);       (* C=48 *)
   C:=ISHIFT(2,6);      (* 2 TO THE POWER OF 6 TIMES 2 *)
   WRITELN(PN,C)        (* C=128 *)
END.
```

```
PROGRAM ISWAPC;
VAR
   RESULT  :INTEGER;
   PN      :TEXT;
BEGIN
   RESET(PN,'PR:');
   RESULT:=ISWAP(1);     (* BIT VALUE 1=0000000000000001 *)
                         (* BIT VALUE FOR SWAP 1=0000000100000000=256 *)

   WRITELN(PN,RESULT);   (* RESULT=256 *)
   RESULT:=ISWAP(3);     (* BIT VALUE 3=0000000000000011 *)
                         (* BIT VALUE FOR SWAP 3=0000001100000000=768 *)

   WRITELN(PN,RESULT);   (* RESULT=768 *)
   RESULT:=ISWAP(2);     (* BIT VALUE 2=0000000000000010 *)
                         (* BIT VALUE FOR SWAP 2=0000001000000000=512 *)

   WRITELN(PN,RESULT)    (* RESULT=512 *)
END.
```

## IXOR Function

Function: Performs a bitwise exclusive OR.

Definition: FUNCTION IXOR(<VAL1>,<VAL2>:INTEGER):INTEGER;

Calling Format: IXOR(<VAL1>,<VAL2>);

Arguments: VAL1 and VAL2 are INTEGER;

Use: IXOR is used for bitwise exclusive oring of INTEGER values.

Example:

```
PROGRAM IXORC;
VAR
   RESULT  :INTEGER;
   PN      :TEXT;
BEGIN
   RESET(PN,'PR:');
   RESULT:=IXOR(1,2);    (* BIT VALUE 1=00000001, 2=00000010 *)
                         (* 1 XOR 2=00000011=3 *)
   WRITELN(PN,RESULT);   (* RESULT = 3 *)
   RESULT:=IXOR(5,14);   (* BIT VALUE 5=00000101, 14=00001110 *)
                         (* 5 XOR 14=00001011=11 *)
   WRITELN(PN,RESULT);   (* RESULT = 11 *)
   RESULT:=IXOR(28,27);  (* BIT VALUE 28 = 00011100, 27=00011011 *)
                         (* 28 XOR 27=00000111=7 *)
   WRITELN(PN,RESULT)    (* RESULT = 7 *)
END.
```

# SECTION 13

# SYSTEM PROGRAMS AND CSS-FILES

# SECTION 13
# SYSTEM PROGRAMS AND CSS-FILES

## 13.1 INTRODUCTION

The PASCAL compiler generates a pseudo-machine code (called p-code). An interpreter is needed to interpret the p-code into machine code so the program can be executed. There are two interpreters available:

- PASSYS - to execute PASCAL system programs.
- PASCAL - to execute user programs.

The following system programs, commands, and modes are also available:

- PASCOMP - to compile PASCAL text files.
- PASCROSS - to create a cross-reference list.
- PASDEL - to delete files.
- PASDUMP - to dump files
- PASLIB - to create and update a PASCAL P-code library.
- PASLINK - to link precompiled text files or library modules into an executable output file.
- PASOBJ - to interpret PASCAL P-code into object code.
- PASPRINT - to list text files
- CSS Mode - to instruct the interpreter to execute CSS commands in a user-specified program file.

Each of the above is described in detail in this section.

## PASCAL Interpreter

**Function:** Interprets p-code user programs into machine code and then executes these programs.

**Format:** PASCAL|,options||,memory| <fd>

**Arguments:** Both fields of the arguments are optional. If either is used, a comma must separate it from the word PASCAL. A second comma must precede the memory field if it is used.

The options can be any letter of the alphabet. If a system program is used, some letters have special meaning depending on the program. If a user program is executed, any letter may be used but its meaning must be defined within the program. The letters have no inherent meaning.

Extra memory may be needed if the program is very large. It can be specified in bytes in the memory field.

The fd is the file descriptor (see Section 1.4).

**Use:** The PASCAL-interpreter is usually used to execute user programs though it can execute all System programs except the compiler.

**Note:** See the PASSYS-interpreter for the list of System programs.

Examples:

PASCAL DATA:NEXTFILE

(Executes the program in the file NEXTFILE on the DATA disk.)

PASCAL,A,1500  DATA:NEW

(Executes NEW with the Abort switch set and 1500 bytes of extra memory.)

PASCAL,,20000  DATA:SEGEXT

(Executes SEGEXT with 20000 bytes of extra memory.)

## PASSYS Interpreter

Function: Interprets p-code System programs into machine code and then executes these programs.

Format: PASSYS|,options| |,memory| <system programs>

Arguments: Both fields of arguments are optional. If either is used, a comman separates it from the word PASSYS. A second comma must precede the memory field if it is used.

The options that are available depend on the System program being executed and are detailed with those programs. The options are not separated from each other by commas.

Extra memory may be needed if a program is very large. It can be specified in the memory field.

The following System programs can be used. They are detailed in the pages that follow:

- PASCOMP - PASCAL Compiler.
- PASCROSS - PASCAL Cross Reference.
- PASDEL - PASCAL Delete File.
- PASDUMP - PASCAL Dump File.
- PASLIB - PASCAL P-code Library.
- PASLINK - PASCAL P-code Linker.
- PASPRINT - PASCAL Print File.

Use: PASSYS is usually used to execute the System programs.

Note: The PASCOMP, PASLIB and PASLINK programs will probably be used the most. The other programs replicate System commands that are unrelated to the PASCAL package such as DEL which deletes files.

PASCOMP is the default program for PASSYS so it can be executed using:

```
PASSYS ,<fd>
```

See PASCOMP for greater detail.

Example:

```
PASSYS,LNI,1500 PASLIB,DATA:KWLIB
```

(Executes the Library program and adds 1500 bytes of additional memory.)

```
PASSYS,,20000 ,DATA:PROGRAMFILE
```

(Executes PASCOMP by default with an additional 20000 bytes of memory.)

## PASCOMP System Program

**Function:** Compiles a PASCAL text file and generates an executable p-code file.

**Format:** PASCOMP,<fd>|,arguments|

**Arguments:** The fd is the file descriptor of the source file to be compiled. Its type should be "ASC" or "ASCPAS".

There are two possible arguments, both of which are optional. A destination file can be specified for the p-code that is generated. The default file is the source file with filetype "BINPAS". Also, a list file with "TEXT" filetype can be specified if the "L" option is used. Its default value is "PR:". A comma must follow the source file if either argument is specified and a second always precedes the list file.

**Use:** PASCOMP is the default program for PASSYS so it does not have to be specified.

The following options are available with PASCOMP:

L - generate a list file and output on the list file descriptor.
E - generate a listing of syntax errors only.
G - allow GOTO statements in the source text. Default value is on.
O - perform I/O check.
R - perform range check.
D - insert line numbers in code file. This will significantly increase the size of the output file.
B - generates additional information for the linker.

Examples:

Ex. 1

```
PASSYS PASCOMP,DATA:KWFILE
PASSYS ,DATA:KWFILE
```

(Both compile KWFILE.)

Ex. 2

```
PASSYS,LRD PASCOMP,DATAT:KWINT,DATA:COMPINT,CON:
```

(Compiles KWINT using "L","R", and "D" options, places the p-code into COMPINT, and outputs the listing to the console.)

Ex. 3

```
PASSYS,LRD ,DATA:KWINT,,CON:
```

(Does the same thing as Ex. 2 except that the p-code is put into KWINT with file type 'BINPAS'.)

## PASCROSS System Program

**Function:** Creates a cross-reference listing of all standard functions and procedures in a program.

**Format:** PASCROSS,<fd>|,fd|

**Arguments:** The first fd (see Section 1.4) references an ASCII source file. The second refers to the destination file. Its default value is "PR:".

**Use:** PASCROSS is used to help locate a program's standard functions and procedures. It is invoked most often for debugging purposes.

**Note:** The entries in the listing are alphabetized with the line numbers on which they appear listed at the right. A line number appears at most once after each entry regardless of the number of times it appears on a line.

The following options are available:

L - add a listing of the program (includes line numbers).

R - include reserved words in the cross reference listing.

F - force a list output in case of end-of-memory.

**Example:** PASSYS,LR PASCROSS,DATA:QUEFILE/A

(Will output the list with the reserved words and the program listing.)

## PASDEL System Program

Function: Deletes files.

Format:
```
PASDEL,<fd>
  or
PASDEL,CMD=<cfd>
```

Arguments: The fd is the file descriptor (see Section 1.4) of the file to be deleted. The default type is 'BINPAS' and any other type may be specified.

The cfd is the file descriptor of the command file that specifies the files to delete. See CSS files in this section for more information.

Use: The command is used to delete files from the Master File Directory.

Note: If the option "D" is placed in the PASSYS option field the command file is deleted on exit.

Examples:

Ex. 1

```
PASSYS  PASDEL,DATA:USELESS/A
```

(Deletes the ASCII type of the file USELESS on volume DATA.)

Ex. 2

The command file (CMDDEL) is:

```
DATA:USELESS
DATA:USELESS/A
DATA:USELESS&/A
$EXIT
```

```
PASSYS  PASDEL,CMD=DATA:CMDDEL
```

(Deletes all three forms of the file USELESS.)

## PASDUMP System Program

**Function:** Dumps the contents of a file.

**Format:** PASDUMP,<fd>|,fd|

**Arguments:** The first file descriptor (see Section 1-4.) is the source file. Its default file type is 'BINPAS'. The second file descriptor is the destination file whose default type is 'PR:'.

**Use:** The command is used to display the contents of a file.

**Note:** The file is outputted in groups of two rows with the character representations above each group. A column in a group represents a byte.

The output is further divided into sets of 256 bytes which constitute a full record. Partial records can also be outputted.

**Example:** PASSYS PASDUMP,DATA:QUEFILE
(Dumps file QUEFILE from volume DATA.)

## PASLIB System Program

**Function:** Creates and updates a PASCAL p-code library that holds pre-compiled functions and procedures.

**Format:** PASLIB,<libfd>|,arguments|

<arguments> "=" <source>
or
<source>,<,lfd>

<source> :=: <fd>|/identifier|
or
CMD = <Cfd>

**Note:** The procedures/functions must be written in the main body of the program via a "PROGRAM EXTERNAL" statement. The procedures/functions added to the library must not contain references to external variables or subroutine calls. Note that "source" must be compiled using switch "B".

**Arguments:** The libfd is the file descriptor (see Section 1-4) of the p-code library. If it is being created, the "N" option must be included in the PASSYS option field.

The lfd is the file descriptor for the list file. IT should be type TEXT and "PR:" is its default value.

The fd in source is the file descriptor of the file where the procedure or function exists. The identifier is the procedure/function name to be added to the library. It can be at most eight characters long.

The Cfd is the file descriptor (see Section 1-4) of the command file that lists the procedures/ functions to be added must be listed in the command file in the file descriptor format specified in Section 1-4. Note type is optional.

There may not be any comments within the list of additions. For more information about command files, see CSS-files in this section.

Use: The library contains subprograms that can be accessed as external subroutines by many different programs.

Note: If no source file is given, the current contents of the library will be outputted to the lfd.

The following options may be used immediately after PASSYS (e.g., PASSYS,<option> PASLIB):

L - generate a listing.
N - create a new library.
I - insert two lines of general information after each procedure is added.

A program can access a procedure in a library by declaring it EXTERNAL. The procedure is then linked with the main program using PASLINK. The format for accessing the subprogram is:

LIB <libfd>

Libfd is the file descriptor (see Section 1-4) of the p-code library.

Examples:

Ex. 1

```
PASSYS,N PASLIB,DATA:KWLIB,DATA:KWPROC(ADD)
```

(Creates a new library, KWLIB, and inserts the subroutine ADD from the file KWPROC. ADD is a subroutine in KWPROC.)

Ex. 2

```
PASSYS PASLIB,DATA:KWLIB,,CON:
```

(Outputs the contents of the library, KWLIB, to the console.)

Ex. 3

DATA:CMDFILE is:

```
  DATA:MATHFILE/SUBVARS
  DATA:MATHFILE/AVERAGE
  DATA:KWSTR/STRMANIP
  $EXIT
```

```
PASSYS PASLIB,DATA:KWLIB,CMD=DATA:CMDFILE
```

(Adds the three subroutines in CMDFILE to the library.)

## PASLINK System Program

**Function:** Links pre-compiled PASCAL text files or library modules into an executable output file.

**Format:** PASLINK,|,CMD=cfd|

**Arguments:** Cfd is the file descriptor (see Section 1-4) of the command file that contains the procedures and segments to be linked together. The default command file is the console.

**Use:** PASLINK must be used to link programs that have segments or external files.

A program can be broken up into segments, and the segments into functions or procedures. These segments are left on disk and brought into main memory only when they are needed. Segments are declared by inserting the word SEGMENT before a procedure declaration.

Example:

```
SEGMENT PROCEDURE procname(....);
```

It is possible to declare segments or procedures/ functions external to a program. They are compiled separately and linked into the main program. A procedure or segment is declared external by appending the word EXTERNAL to the procedure or segment declared.

Examples:

```
PROCEDURE elsewhere(....);
  EXTERNAL;

SEGMENT PROCEDURE onward(....);
  EXTERNAL;
```

The actual procedure is defined in another file; only the heading appears in the main program. However, the heading must appear exactly as it does in the file in which the procedure is defined except the words SEGMENT or EXTERNAL.

If the external procedure or segment performs any I/O, an output file must be passed to it from the main file. The format is:

```
VAR <iofd> : TEXT
```

The I/O file must be used for all output to the console that is performed in the external procedure/segment. For example:

```
WRITELN(<iofd>,'IN SEGMENT')
```

Iofd (see Section 1-4) would be the same as the one the hedefined in the heading. Failure to include will the output file will result in errors when the files are linked.

There are certain rules that must be followed when functions, procedures and segments are declared external:

1. The main program may contain external segments and procedures.

2. All segments must be declared in the main program.
3. An external segment may contain external procedures but an external procedure may not.
4. Declarations must appear in the following order:

   External procedures and segments
   Internal segments
   Internal procedures

   The procedures may be declared FORWARD if necessary.

Values are passed to external procedures and segments through the parameters in the headings.

Procedures that are declared external must be defined in a separate module, the general outlay of which should be:

Program heading: The word PROGRAM should be followed directly by the word EXTERNAL. There is no program name.

Global variable declaration: All global variables used by the procedure(s) in the module should be listed using the normal variable declaration format. The global variables must be defined exactly the same way in the main program.

External procedure and function declaration: Any procedures or functions that are used by the procedure(s) in the module but are external to the module are listed in the same way they are in the main program. Procedures that are in a different segment must be declared as external segmented procedures.

Procedure declaration: All local procedures are then listed. All local variables are listed inside of the specific procedures that use them. External procedures can be declared within these procedures and they are defined in a separate module. It is important that all local procedures, external or nonexternal, are defined, even if they are only used by external procedures.

Empty main program: There is no real "body" of the program - only the words "BEGIN END".

Note that several external procedures may be defined in the same module. Also, all procedures, internal and external, and global variables must have unique names.

<u>Example</u>:

```
PROGRAM    EXTERNAL;
   VAR    G1,G2,G3 : SOME TYPE    (* global vars *)

   PROCEDURE  GETCHAR(...);
      EXTERNAL;     (* external procedure *)

   SEGMENT PROCEDURE GETTOK (...);
      EXTERNAL;     (* external segmented procedure *)

   PROCEDURE TOKEN(...);     (* local procedure *)

      PROCEDURE PRINTOUT(...);
         EXTERNAL;   (* external procedure referenced by TOKEN *)

      PROCEDURE LOCAL
         BEGIN  (* body of LOCAL *)
           :
         END;

      BEGIN
        GETCHAR(...);
        GETTOK(...);    (* body of TOKEN *)
        LOCAL;
        PRINTOUT;
      END;

   BEGIN   END.  (* empty main body *)
```

Procedure GETTOK would look like this:

```
PROGRAM EXTERNAL;

(* global variables and external procedures *)

  SEGMENT PROCEDURE GETTOK(...);
     BEGIN  (* body of GETTOK *)
       :
     END;

  BEGIN   END.
```

There are no restrictions as to where procedures, segmented or non-segmented, may occur. However, all procedure and global variable names must be unique.

All modules must be compiled with the switch "B" so that the compiler will generate information used by the linker.

Linker Commands

Linker commands are executed in command (CSS) files. The following commands are available to be used with the linker:

Note: In the commands below, fdname is the file descriptor (see Section 1-4) for the PASCAL p-code file containing the external segment or procedure being linked:

1. INC, <fdname>
   Includes all procedures found in "fdname". Procedures in FNAME that are not currently referenced in the program will be included as global procedures.

2. LIB|,R| <fdname>
   Includes only those procedures in "fname" that are currently referenced in the program but have not yet been included. If the option R (REPLACE) is used, any procedure in future that is already included in the program will be deleted and the new version brought in from "fdname". This command is also used to collect procedures from a library.

3. TASK|,B| <fdname>

   Name the output file "fdname". This command can be given anywhere in the command stream. If the option B is given the output file will include linker information.

4. PRINT|,M| <fdname>

   A listing will be sent to "fdname". If the option M is given, a program layout of all procedures will be included in the list file.

5. CHECK

   Gives a list of all procedures which are currently referenced but not yet included.

6. ABORT

   Abort the linker.

7. END

   Finish the linking session.

When linking the external modules the following order of commands should be used:

1. Include the main program using the INC-command.
2. Include all global non-segmented procedures declared in the main program.
3. Include all global segmented procedures declared in the main program.

Collect all procedures which are local to procedures included so far. If the procedures included contain local external procedures the INC-command should be used. If not the LIB-command should be used. Repeat this procedure until all external procedures have been included.

Note that global procedures which are not referenced in the main program need not be declared provided these procedures are included, using the INC-command, immediately following the INC-command of the main program.

It is very important that the declaration heading of the procedure being linked is absolutely identical to the declaration heading where the procedure is declared external.

Example: in DATA:MAINFILE:

```
PROGRAM mainprogram:

  PROCEDURE procext(VAR io : TEXT);
   EXTERNAL;

  PROCEDURE printext(VAR io : TEXT);
   EXTERNAL;

  SEGMENT PROCEDURE segext(VAR io : TEXT);
    EXTERNAL;

  SEGMENT PROCEDURE segint;
    BEGIN
     WRITELN('IN segint')
    END; (* segint *)

  PROCEDURE procint;
    BEGIN
     WRITELN('IN procint')
    END; (* procint *)

  BEGIN
     procext(OUTPUT);  (* output file for an
                          external procedure *)
     segint;
     printext(OUTPUT);
     segext(OUTPUT);
     procint;
     WRITELN('FINISHED')
  END.  (* mainprogram *)
```

in DATA:PROCEXTFILE:

```
PROGRAM EXTERNAL;

   PROCEDURE procext(VAR  io : TEXT);
     BEGIN
       WRITELN(io, 'IN procext')
     END;  (* procext *)

   PROCEDURE printext(VAR  io : TEXT);
     BEGIN
       WRITELN(io, 'IN printext')
     END

   BEGIN
   END.  (* dummy *)
```

in DATA:SEGEXTFILE:

```
PROGRAM EXTERNAL;

  SEGMENT PROCEDURE segext(VAR  io : TEXT);
    BEGIN
      WRITELN(io, 'IN segext')
    END;  (* segext *)

  BEGIN
  END.  (* dummy *)
```

in DATA:CMDFILE:

```
INC      DATA:MAINFILE     Include the main program
INC      DATA:PROCEXTFILE  Include the procedure
LIB      DATA:SEGEXTFILE   Get the segmented procedure
PRINT,M  LISTFD            Define list file
TASK,B   TASKFD            Define output file
END                        End of commands
```

To execute the program:

1. compile:

```
PASSYS ,B DATA:MAINFILE
PASSYS ,DATA:MAINFILE
PASSYS ,DATA:PROCEXTFILE
PASSYS ,DATA:SEGEXTFILE
```

2. link:
```
PASSYS PASLINK, CMD=CMDFILE
```

<u>or</u>

```
PASSYS PASLINK,CMD=CON:
(type in all the statements in CMDFILE in
order)
AB
```

3. execute:
```
PASCAL DATA:LINKED
```

output:
```
IN procext
IN segint
IN printext
IN segext
IN procint
FINISHED
```

'INC DATA:MAINFILE' includes the main program.

'INC DATA:PROCEXTFILE' includes both procedures in PROCEXTFILE.

"LIB DATA:SEGEXTFILE' brings in the segmented procedure "segext". INC could have been used instead. Care should be taken when the INC-command is used with a p-code library since all procedures in the library will be included. LIB should be used since this will only bring in those procedures that have been referenced.

## PASOBJ System Program

**Function:** Interprets PASCAL P-code into object code.

**Format:** PASOBJ <infd>|,outfd|

**Arguments:** Both arguments are file descriptors (see Section 1-4): the first is the file descriptor that contains the PASCAL P-code and the second is the one into which the relocatable object code will be placed. The default for outfd is infd with type "OBJ".

**Use:** PASOBJ is required for the user to create a task file. Tasks are often preferred to P-code because they execute extremely fast.

Note: PASOBJ is not executed in conjunction with PASSYS.

Although the creation of the object code file is crucial, it is only one part of the conversion from a PASCAL program to a task. First, the file must be compiled. Then it is converted into object code using PASOBJ. Last, the task establisher, RLDR, is called to perform the final conversion. Its format is:

RLDR|,switches|,mem|| CMD = <commandfile>

The following switches are available:

R - Additional code for range checking is generated.
O - Additional code for I/O-checking is generated.

MEM is extra memory that may be allocated. The commandfile may be a CSS-file or "CON:". Regardless of whether or not the file is interactive, it must consist of the following commands:

| | |
|---|---|
| \|LOG CON:\| | (Displays each comand as it is executed-not used in interactive files.) |
| OPTION NOSTACK | (No stack check is performed.) |
| \|STACK XXX\| | (Expand the stack by XXX bytes.) |
| INC <PASOBJ fd> | (Include as many object files as needed.) |
| ⋮ | |
| LIB PASRTL | (Collects modules from the PASCAL Runtime Library.) |
| Task <outfd> | (Links the objectfiles into outfile.) |
| END | (Terminates RLDR.) |

All variables in the program are pushed onto the stack at run-time so the stack may need to be expanded since it starts with only 256 bytes. If the task terminates with an End-of-Memory error, the program should be relinked with a larger argument in the STACK-command.

The following limitations must be considered when PASOBJ is used in conjunction with PASCAL programs:

1. The sourcefile may not contain segmented procedures or functions.

2. The function EXIT(NAME) is allowed provided NAME is the name of a procedure and not a function.

Example: (TEST is a PASCAL file)

in CMDFILE

```
LOG  CON:
OPTION NOSTACK
STACK 500
INC  TESTOBJ
LIB  PASRTL
TASK  TEST TASK
END
```

To create the taskfile TEST TASK:

```
PASSYS ,TEST          (compiling TEST)
PASOBJ TEST,TESTOBJ   (objectcode is placed in
                      TESTOBJ)
RLDR,RO,20000 CMD=COMDFILE (create the task)
```

To execute the task:

```
TEST TASK
```

(RLDR is described in detail in Appendix L.)

## PASPRINT System Program

Function: Gives a list output of the specified text file.

Format: PASPRINT,<fd>|,fd|

Arguments: The first fd is the file descriptor (see Section 1-4) of the source file. It must be a text or list file. The second fd is the file descriptor of the destination file whose default type is 'PR:'.

Use: This command is used to output file listings and text files.

Note: If the file contains a listing of a program, it will be printed with line-numbers.

If the "I" option is placed in the PASSYS option field no line-numbers or form feeds will be generated on output. This option is designed to be used with list files.

Example: PASSYS PASPRINT,DATA:QUEFILE
(print QUEFILE's listing with line-numbers.)

## 13.2 CSS-MODE

The user can instruct the interpreter to execute commands or to accept input by executing in CSS-mode (Command String Supervisor). The commands it executes are in a CSS-file that is user-defined like a PASCAL program file except that it is not compiled.

The command file is initially created using the Editor the same way that PASCAL files are formed. Any command that can instruct the interpreter in non-CSS-mode can be used in CSS-mode.

Any command-line that starts with the character "*" is a comment and is ignored by the CSS-processor. The $EXIT-command informs the interpreter that no more commands are available and terminates the CSS file.

The files are often executed directly by PASSYS by placing a slash ('/') before the CSS-filename.

Example: PASSYS/DATA:CMDFILE
PASSYS/CON: (for interactive mode)

CMDFILE is a CSS-file holding commands for the CSS-processor. The command can start the execution of a CSS-file using the format:

/<cfd>|,parameters|

Example: PASSYS/DATA:CMDFILE,val1

The parameters that are passed to a CSS-file can be accessed in the program number ("1" through "9" determined by the order in which the parameters are listed). If these two symbols are found in a command, the corresponding parameter will be automatically supplied.

Example: PASDEL,@1

The file with the first parameter's name and of type 'BinPas' would be deleted when this command was read.

When passing start switches to a PASCAL program, the switches must be inclosed by the characters '<' and '>'. Note that in all other areas of this manual these characters enclose required fields and are never typed. A global switch 'Q' has been implemented.

Format: <switches> <PASCAL System Program>,<fd>

Example: <Q>PASCOMP,TEXTFILE
Starts compiler, Q inhibits all console output.

$-Commands

The following $-commands define the current CSS-mode and are interpreted by the CSS-processor:

1. $LOGG

   All CSS-commands will be logged on the system console before they are executed.

2. $NOLOGG

   The CSS-commands are not logged on the console before execution. The system default is $NOLOGG.

3. $TEST

   A HALT instruction is implicitly executed when a fatal programming error is detected. It can also be used as a PASCAL command by the user.

4. $REMOTE

   If a program being executed in CSS-mode executes a HALT, whether implicitly or explicitly, the CSS will go to the End-of-Task. $REMOTE is the default value.

5. $EXIT

   Execution of the current CSS-file is terminated. If the files are nested, the outer files are not terminated.

6. $PURGE

   Execution is terminated on the current CSS-file and the file is deleted.

7. $HALT

   Stops all CSS-mode execution, regardless of the number of nested program levels, and goes to End-of-task.

$$-Commands

The $$-commands are powerful CSS-commands that allow more sophisticated programming in the CSS-mode. These commands are interpreted by the system program PASCSS which must be present on the system volume. The syntax for a $$-command is:

    $$|label:|<;command>

Several $$-commands may be written on the same line. The commands are separated by the character ";". The last command on a line may be of any type.

Example:

    $$LOAD TASKA; $$START TASKA; $E

The label is any integer between 1 and 99. The command can be any CSS-command.

Flow of Control and Execution Commands

The following are the $$-commands that determine the flow of control and execute internal portions of the CSS-command file:

1. $$GOTO <label>
   The command will force a change in the flow of control from the current command to the one preceded by the label.

2. $$DISPLAY <string>
   The string is written to the console.

3. $$SLEEP <n>
   CSS-execution is suspended for n seconds. n must be a positive integer.

4. $$IF <condition>

   .

   .

   .

   $$ENDI

   The $$IF-command executes the same way the PASCAL IF-statement does, i.e. if the condition is True, the CSS-processor executes the CSS-commands following the $$IF-command. Otherwise, it executes the command following the matching $$ENDI (upper case only)-command. The possible conditions are:

   Par(p) True when p is a string.
   NOPAR(p) True when p is the null-string.

   PAR(p1=p2) True if p1 equals p2.
   NOPAR(p1=p2) True if p1 does not equal p2.

   TASK(tid) True if the task tid is present in memory. (tid is the task fd.)
   NOTASK(tid) True if tid is not in memory.

   FILE(fd) True if the specified file exists. Default in ASCII. Any other file type requires "/type" to be appended to the filename.
   NOFILE(fd) True if the specified file does not exist or the file is open.

   ERROR True if the previous PASCAL program executed a HALT-instruction. This condition implies the execution of a $TEST-command.
   NOERROR True if the preceding PASCAL program did not execute a HALT-instruction. More than one condition can be tested for using the following format:
   $$IF <condition1>,<condition2>|,condition,...|

Example:

```
$LOGG
* sample CSS-file using some of the commands
* described on the preceding pages.
$$IF PAR(@1=TEST)
$$DISPLAY @1
$TEST
$$GOTO 1
$$ENDI
$REMOTE
* compile TESTFILE
$$ 1:,TESTFILE
* execute TESTFILE using PASSYS
TESTFILE
$$DISPLAY 'END OF CSS-FILE'
$EXIT
```

5. $$RUN <fd>|,parameters| or $$RUN </cssfile>|,Parameters|
The program fd is started. fd may be either a PASCAL file or a CSS-file (requires slash before name). The parameters are optional and may be a list of starting parameters, a select file containing a set of starting parameters or both. The select filename is preceded by 'SEL='. There may be at most one select file and each line in the file contains one set of parameters. The CSS-processor will repeat the execution of fd until all the parameters in the select file have been used, i.e., fd will be executed at least once for every line in the select file. The $$RUN command may not be nested and must not be followed by a command on the same line.

Example: in PFD:

```
SEGMENTTEST,NRMLEXT,SEG1EXT
```

in CSS-file:

```
$$RUN /LINKING,SEL=PFD,CMDFILE
```

LINKING is a CSS-file that invokes PASLINK. CMDFILE is just an additional parameter. The parameters are all accessed using the @ character, i.e., CMDFILE would be @4.

6. $$ACTIVATE |parameters|

⋮

$$ENDA (Enter $$ENDA in upper case only)

The CSS-commands between the two commands above are formed into a temporary CSS-file and activated. The parameters are optional and may be a list of starting parameters, a select file, or both. Each parameter is delimited from the others by commas, both in the parameter list and in the select file. At most, one select file may be specified in a parameter list. The temporary CSS-file will be executed repeatedly until all the parameters in the select file are used, i.e., once for every line in the select file since all the parameters for one execution are listed on one line. The $$ACTIVATE command may not be nested and may not be followed by another command on the same line.

Example: (using the same PFD as on previous page)

```
$$ACTIVATE  SEL=PFD,CMDFILE
/LINKING, @1,@2,@3,@4
$$DISPLAY 'LINKED'
$$ENDA
    ⋮
```

Taskfile Commands

Certain CSS-commands operate on task files. They may be user-developed or system tasks. In each command listed below, tid is a four-letter name assigned to the task when it is loaded into memory.

1. $$LOAD <fd>|,arguments|

Arguments :=: tid,Mem,R

The task in the field is loaded into memory under the name tid. If tid is not specified, the first four letters of the fd file descriptor is used. Additional memory may be allocated using the Mem field. The memory amount is in bytes. if the letter 'R' is specified, the task will remain resident in memory until it is removed. (See

$$KILL.) If the task is resident in memory, it cannot be loaded in again. If any of the arguments are used, all the commas must be included.

2. $$START <tid>|,switches|,priority||parameters|
   The task that exists in memory specified by tid is started with any switches, priority, or starting parameters that are needed.

3. $$PAUSE |tid|
   The task specified is paused. (Default is CSS.)

4. $$CONTINUE <tid>
   The task specified by tid is continued.

5. $$WAIT <tid> CSS is suspended until the task specified by tid has completed execution and gone to End-of-Task.

6. $$TRIGG <tid>
   CSS is suspended until the task specified by tid changes states. Note: If tid has not been started or has already completed, the system must be rebooted and the files that were left open closed using DISKCHECK.

7. $$KILL <tid>
   The task specified by tid is cancelled and removed from memory.

8. $$PRIORITY <tid>,<priority>
   The task tid, is assigned a priority between 1 and 255.

9. $$OPTION <tid>,<opt1>,<op2>
   The options of task tid is changed to op1, op2, etc. where:

| Option | | Meaning |
|---|---|---|
| AB | - | Task abortable. |
| NAB | - | Task not abortable |
| RES | - | Task is memory resident. |
| NRES | - | Task is not memory resident. |

## Creation of Permanent Files

The following commands create, destroy, and rename permanent files from within a CSS-file:

1. $$BUILD <cfd>|,parameters|

   ⋮

   $$ENDB

   The CSS-commands between the two listed above are formed into a CSS-file and named cfd. The newly created file will be type "AscPas" so it cannot be edited though it can be executed. The parameters are optional. Even if they are omitted when the file is created, they can be included when it is executed. The $$BUILD-comand may not be nested and may not be followed by a command on the same line.

2. $$FILE <fd>|,parameters|

   ⋮

   $$ENDF

   The lines between the commands are formed into a file and named fd. The files may contain either PASCAL or CSS-commands. The file can be executed but not edited because it is type "AscPas". The parameters are optional but if they are omitted when the file is created, they cannot be included later. This is one of the differences between $$FILE and $$BUILD. $$FILE commands may not be nested and may not be followed by a command on the same line.

3. $$DELETE <fd>

   The file specified by fd is deleted. This is especially important since a new file cannot be created by either of the above commands if it already exists.

4. $$RENAME <fd1>,<fd2>

   File fd1 is renamed fd2. Files other than ASCII require the type to be specified.

$$-Commands in Interactive Mode

It is possible to run the $$-commands in an interactive mode. Characters enclosed in apostrophes are displayed on the console and a prompt is output. The characters entered by the user then replaces the original string before the command is activated.

Example:

```
$$IF  PAR('CONTINUE(Y/N)?'=N)
 $HALT
$$ENDI
```

In this case the string 'CONTINUE(Y/N)?' followed by a prompt is displayed on the console. If the user enters the character 'N', this character replaces the entire string, as shown below.

```
$$IF  PAR(N=N)
$HALT
$$ENDI
```

Since the condition is true the $HALT command will be executed.

## Illustrated Example

```
********************************************************************
*                                                                  *
*                                                                  *
*          THIS IS AN EXAMPLE OF A CSS-FILE WHICH                  *
*          CREATES AN EXECUTABLE TASK FILE FROM                    *
*          A PASCAL TEXT FILE.                                     *
*                                                                  *
*          THE CSS HAS FOUR INPUT PARAMETERS:                      *
*                                                                  *
*          P1 = PASCAL TEXT FILE.                                  *
*          P2 = NAME OF TASK FILE        ( DEFAULT:  P1         )  *
*          P3 = ADDITIONAL STACK SIZE    ( DEFAULT:  2000 BYTES )  *
*          P4 = LIST FILE FOR COMPILER   ( DEFAULT:  NULL:      )  *
*                                                                  *
*                                                                  *
*                                                                  *
********************************************************************
*
*
*
*          HI THERE...
*
$$DISP;$$DISP Create Pascal Task  -  CSS  1.00
*
*
*
*          PARAMETER 1 MUST BE SPECIFIED!
*
$$ IF NOPAR(@1);$$FILE CON:
Parameter error!
Enter: source-fd,<task-fd>,<task stack-size>,<list-fd>
Default: task-fd=source-fd, task stack-size=2000
$$ENDF;$E;$$ENDI
*
*
*
*          MAKE SURE RECOURCES ARE AVAILABLE...
*
$$IF FILE(RLDR/T),FILE(PASOBJ/T),FILE(PASRTL/O);$$GOTO 10;$$ENDI
*
*          IF NOT, THIS IS WHAT WE NEED!
*
$$FILE CON:
This CSS requires the following files on the system volume:
  RLDR   - Task establisher.
  PASOBJ - Pascal object code generator.
  PASRTL - Pascal run time library
$$ENDF;$E
*
*
```

Illustrated Example (Cont.)

```
*
*          NO USE TO INVOKE THE COMPILER IF PARAMETER 1 IS IN ERROR!
*
$$10:IF NOFILE(@1/A)
  $$DISP Assign error on source file @1!;$E
$$ENDI
*
*
*
$TEST
*CSS WILL NOT BE ABORTED ON ERRORS!
*
*
*          COMPILE SOURCE FILE, PARAMETER 4 IS LIST FILE!
*
$$DISP Compilation started...;<Q>PASCOMP,@1,CSS%B,@4
*
*
*
*          IF COMPILATION FAILED - STOP CSS!
*
$$IF ERROR
  $$DISP Compilation error on source file @1!;$E
$$ENDI
*
*
*
*          CREATE THE OBJECT FILE
*
$$LO PASOBJ,PO;$$PRI PO,200;$$ST PO CSS%B;$$WAIT PO
*
*
*          DELETE P-CODE TEMP FILE
*
$$DEL CSS%B/B
*
*
*
*          BUILD A CSS WHICH DOES THE LINKING STUFF...
*
*          --------------- START OF CSS FILE ---------------
$$BUILD CSS%LINK
*
*
*          THIS CSS HAS TWO PARAMETERS:
*
*          P1 = TASK FILE
*          P2 = STACK SIZE
*
*
```

Illustrated Examples (Cont.)

```
*
*           BUILD A COMMAND FILE FOR RLDR
*
*           ++++++++++  COMMAND FILE FOR RLDR  ++++++++++
$$FILE CSS%C
OPTION NOSTACK                      NO STACK CHECK!
STACK @2                            STACK SIZE IS A PARAMETER!
INC CSS%B                           INCLUDE TEMP OBJECT FILE
LIB PASRTL                          RUN TIME LIBRARY...
TASK @1                             TASK FILE IS A PARAMETER!
END
$$ENDF
*           ++++++++++  END OF COMMAND FILE  ++++++++++
*
*
*           START RLDR AND PRODUCE A TASK FILE
*
$$LO RLDR,,10;$$PRI RLDR,200;$$ST RLDR CMD=CSS%C;$$WAIT RLDR
*
*
*
*           DELETE OBJECT FILE AND COMMAND FILE FOR RLDR
*
$$DEL CSS%B/0;$$DEL CSS%C/A
*
*           ALL IS OK!!
*
$$DISP;$$DISP Task @1 created!
*
*
$PURGE      CSS FILE WILL BE DELETED ON EXIT!
*
$$ENDB
*           --------------- END OF CSS FILE! ---------------
*
*
*
*           RUN THE CREATED CSS WITH PROPER PARAMETERS
*
$$IF NOPAR(@2),NOPAR(@3);/CSS%LINK,@1,2000
$E;$$ENDI
*
$$IF PAR(@2),NOPAR(@3);/CSS%LINK,@2,2000
$E;$$ENDI
*
$$IF NOPAR(@2),PAR(@3);/CSS%LINK,@1,@3
$E;$$ENDI
*
/CSS%LINK,@2,@3
$E
```

# SECTION 14

# ISAM STATEMENTS

1

SECTION 14

ISAM STATEMENTS

## 14.1 INTRODUCTION

ISAM, Indexed Sequential Access Method, is a technique used for indexed access to large data files. It can be used for random access using a particular key string as the search argument, or sequential access using the index. The ISAM data and index files are initialized by a utility program. After initialization, these files are loaded by the user via ISAM write operations. They can be modified using the other ISAM statements. This section describes the statements used to load and modify ISAM data files.

The task file ISAM must be on the system disk in order to successfully compile and execute a program containing ISAM statements. If no more ISAM programs are to be compiled or executed, utility program "KILLISAM" should be run to release the space occupied by the ISAM task file.

### ISAM Error Handling

When an ISAM error occurs (e.g., key not found condition) during execution of an ISAM PASCAL program, the ISAM Task sends to the PASCAL program the appropriate error code. This error code is contained in IORESULT. In order to display this error code, set IORESULT equal to some variable and then display the variable via WRITELN. For example:

```
X: = IORESULT;
WRITELN(X);
```

The following ISAM error codes can be returned:

| Code | Meaning |
|---|---|
| 120 | ISAM - key not found |
| 121 | ISAM - duplicate key |
| 122 | ISAM - illegal key value |
| 123 | ISAM - mismatch at check-read |
| 124 | ISAM - index not found |
| 125 | ISAM - data record length invalid |
| 126 | ISAM - task: end of memory |

Refer to procedure CHECK of program ISAMDEMO in Appendix G for a sample method of handling ISAM errors.

## 14.2 ISAM CREATE PROCEDURE

Function: Allocates and creates an ISAM Index File and its associated data file.

Format: -CREINDEX (CREINDEX is a task file on your disk.)

Use: To create and allocate ISAM files requires the execution of Utility Program CREINDEX. Refer to 8800 Series "Monroe Utility Programs Programmer's Reference Manual" for important information about ISAM files and procedure instructions for CREINDEX. The information below is taken from this manual and shown here for your convenience.

When program CREDINDEX is executed it prompts the user as follows:

```
Enter name of index file? _____
Preallocate space (Y or N)? _____
Enter name of data file? _____
Preallocate space (Y or N)? _____
Enter record length? _____
Enter key start position? _____
Enter key type (B, A, I, F or D)? _____
Ascending or Descending sequence (A/D)? _____
Are duplicate key values allowed (Y or N)? _____
Are there any more indices (Y or N)? _____
Is information correct (Y or N)? _____
```

If there are any more indices, the user is returned to the first query inputting the name of the index file, the name of the data file, and so on, until all indices have been entered. Then a table is output to the console summarizing all of the information entered during the session. The program terminates after the following questions are answered:

```
Information correct (Y or N)? _____
Would you like a copy on the printer (Y or N)? ____
```

Example:

This example illustrates how an index and a data file are allocated, created and then built. Three indexes are specified: Name, Number and Dept. These files and indexes will be referenced in the ISAM program examples that follow.

In this example the files CREINDEX, RETEST and PERSONNEL are located on a data disk named DATA:.

```
-DATA:CREINDEX¶
**CEATE ISAM FILES  Ver. m.nn***

Enter name of index file?  DATA:RCTEST¶
Preallocate space (Y or N)?  N¶
Enter name of data file?  DATA:PERSONNEL¶
Preallocate space (Y or N)?  N¶
Enter record length?  80¶
Enter name of index?  NAME¶
Enter key start position 1¶
Enter key length?  30¶
Enter key type (B, A, I, F or D)?  A¶
Ascending or Descending sequence (A/D)?  A¶
Are duplicate key values allowed (Y or N)?  Y¶
Are there any more indices (Y or N)?  Y¶
```

The questions are repeated with the following entries:

```
NUMBER¶     30¶     A¶     A¶     N¶     Y¶
DEPT¶       20¶     A¶     A¶     Y¶     N¶
```

The following output appears on the console:

```
***Create Isam Files***     Ver. P-m.nn      xxx-mm-dd/hh-mm.ss

                    Data and Index File Information.

                    Index File name: data:rctest
                    Data File name : data:personnel
                    Record size    : 80

                     Filename      Reclgt   BlkSize   Allo blks
                     --------      ------   -------   ---------
Index File:          RCTEST          256    Default   Default
Data File:           PERSONNEL        80    Default   Default

Index No.  Index Name  Key Type  Sort order  Dupl.  Key Start/Length
---------  ----------  --------  ----------  -----  ----------------
    1      name         Ascii    Ascending    Yes      1/30
    2      number       Ascii    Ascending    Yes     31/30
    3      dept         Ascii    Ascending    Yes     61/20

                    Is information correct (Y or N)? Y¶
                    Would you like a copy on the printer (Y or N)?  N¶
```

The program ends with the message:

```
Index file created!
Data file created!
End of task 0.
```

Loading the Data File

Now, using the index file data:rctest and the data file, data:personnel, a program can be written to input information into the data file as a single string of 80 characters.

This program must include the following statements to open the ISAM index file and load its associated data file:

1. OUTFILE:isamfile;
2. RESET (outfile,'data:rctest');
3. ISAM (outfile,WRITE,strtwo);

Statement 1 "outfile:isamfile;" appears in the variable declaration section of every PASCAL program that uses ISAM.

Statement 2 "RESET(outfile,'data:rctest');" associates the declared filename outfile with the CREINDEX index file name 'data:rctest' which must appear in quotes. Note that the volume name data: refers to the name of the disk where the index file is stored.

Statement 3 "ISAM(outfile,write,strtwo);" is the format of the ISAM write statement. It is explained in detail in Section 14.6.

Sample Program:
Program ISAMDEMO, shown in Appendix G, is one type of program which will load and modify index file DATA:RCTEST just created.

This program is for illustration purposes only; it is not intended to demonstrate the best possible programming techniques.

Program ISAMDEMO can be compiled using the following command:

```
-PASSYS,L,10000 ,DATA:ISAMDEMO,,CON:¶
```

The program is executed as follows:

```
-PASCAL DATA:ISAMDEMO¶
```

The program performs all of the ISAM techniques discussed in subsequent parts of this section. The information you enter into the program is maintained in both index file (DATA:RCTEST) and associated data file (DATA:PERSONNEL). The information entered may then be read, modified, or deleted by invoking the appropriate procedure in the program.

Note that the ISAM data file can be displayed on the console with the following utility command:

```
-COPYLIB DATA:PERSONNEL,CON:¶
```

## 14.3 ISAM DELETE STATEMENT

Function: Removes a particular data record's keys from an ISAM index file.

Format: ISAM(<isamfileid>,DELETE,<destring>);

Arguments: Isamfileid is a string declared in the VAR section of the program or procedure heading and again associated with the ISAM data file in the RESET statement. For example:

Declaration:

```
VAR
 isamfilename:ISAMFILE
```

Main Section:

```
    begin
      :
    RESET(isamfilename,'data:filename');
      :
    ISAM (isamfilename, DELETE, destring);
```

Destring is a string containing the record to be deleted.

Use: This statement removes the appropriate keys from a designated record in the ISAM file. The associated data record is not touched but subsequent access is not possible. Before an ISAM DELETE can be done the record must be ISAM READ (see Section 14.4).

Note: Destination string should be equal to the declared record length.

Example: Refer to the RECDEL procedure of program ISAMDEMO in Appendix G.

## 14.4 ISAM READ STATEMENTS

Function: Accesses by key or sequentially, records contained in the Data File associated with an ISAM Index File.

Format:

1. ISAM(<isamfileid>,READLAST,<destring>
   [,indexstring]);
2. ISAM(<isamfileid>,READFIRST,<destring>
   [,indexstring]);
3. ISAM(<isamfileid>,READPREVIOUS,<destring>
   [,indexstring]);
4. ISAM(<isamfileid>,READNEXT,<destring>
   [indexstring]);
5. ISAM(<isamfileid>,READKEY,<destring>[,indexstring],
   <keystring>);

Arguments: Isamfileid refers to the filename declared in the VAR section of the program and associated with the ISAM data file in the RESET statement.

Destring refers to the string that is loaded with the designated record from the ISAM data file as a result of the ISAM read operation. This string is available for manipulation by PASCAL string statements (e.g., COPY(string,1,2);).

Indexstring refers to an index name set up by Task CREINDEX and stored in the index file. If omitted, the default is the first index except for Format 5 above; for this case, the search starts at the first index and proceeds through the last until the key is found.

Keystring refers to the string containing the specific key to be searched for. If the length of the keystring is greater than the key, the keystring is truncated and matched against the key. If the reverse is true, the key is truncated to the same length as keystring; keystring is then matched against the truncated key.

Use: The particular record that is accessed depends on what keyword is included in the ISAM Read Statement. The available keywords are:

READLAST - reads the last record in logical order for the index specified in indexstring.

READFIRST - reads the first logical record for the index specified in indexstring.

READPREVIOUS - reads the previous record in reverse logical order using the file pointer as the offset for the index specified in indexstring.

READNEXT - reads the next record in logical order using the file pointer as the offset for the index specified in the indexstring.

READKEY - performs a sequential search for the key or subkey string in the specified index. If it finds the record it places it into the destination string. If the index string is omitted, it performs a sequential search for the key or subkey string (starting at the first index proceeding through the last) and reads that record into the destring.

Examples:

```
ISAM(outfile,READKEY,'NAME','JONES')
ISAM(outfile,READKEY,'JONES')
ISAM(outfile,READKEY,'JON')
```

Example: Refer to the RECREAD procedure of program ISAMDEMO in Appendix G.

## 14.5 ISAM UPDATE STATEMENT

Function: Replaces a specified record in the ISAM data file and produces key changes to the index file where appropriate.

Format: ISAM(<isamfileid>,UPDATE,<oldstring>,<newstring>);

Arguments: Isamfileid is a string declared in the VAR section of the program or procedure heading and again associated with the ISAM data file in the RESET statement. For example:

Declaration:

```
VAR
isamfilename:ISAMFILE
```

Main Section:

```
    begin
      :
    RESET(isamfilename,'data:filename');
      :
    ISAM (isamfilename,update,oldstring,newstring);
```

Oldstring is the string corresponding to the record being replaced.

Newstring is the string to be inserted into the ISAM data file in place of the oldstring. All changed indices will be updated when this replacement occurs.

Use: The ISAM UPDATE statement will exchange one record string for another in the ISAM data file. The only restriction is that oldstring and newstring must be of equal length or an update will not occur. Before using this statement, the appropriate ISAM file must be opened (via RESET) and the desired record read via an ISAM Read operation.

If a duplicate key occurs in an index where it is not allowed, that index will not be updated. For example, if the name SMITH was used as a key for record 50 and you wanted to change record 20's key to SMITH, record 20 would not be updated. In order to keep the indices properly updated, an ISAM DELETE operation must be performed.

Example: Refer to Procedure RECUPDATE of program ISAMDEMO in Appendix G.

## 14.6 ISAM WRITE STATEMENT

**Function:** Enters a new record into the ISAM Data File and adds the new keys in the Index File.

**Format:** ISAM(<isamfileid>,WRITE,<isamrecord>);

**Arguments:** Isamfileid is a string declared in the VAR section of the program or procedure heading and again associated with the ISAM data file in the RESET statement. For example:

Declaration:

```
VAR
  isamfilename:ISAMFILE
```

Main Section:

```
begin
  :
RESET(isamfilename,'data:filename');
  :
ISAM(isamfilename,WRITE,isamrecord);
```

Isamrecord is a variable containing the string (record) to be written to the ISAM Data File. The isamrecord must be equal in length to the record length specified when the file was created by Task CREINDEX. If the length of isamrecord is not equal to the declared record length for that associated ISAM data file, no information will be written to the data file or index file.

Note: A good practice is to test the length of isamrecord with the PASCAL string statement LENGTH(isamrecord) to insure the proper execution of the ISAM WRITE statement.

Use: The record is appended to the data file and all indices are updated. The record must contain information in all key fields. If a duplicate key occurs in an index where it is not allowed, that record will not be written.

Example: Refer to Procedure RECWRITE of program ISAMDEMO in Appendix G.

# APPENDIX A

# QUICK REFERENCE SUMMARY

1

## APPENDIX A
## QUICK REFERENCE SUMMARY

**Reference & Format** — **Use** — **Page**

ABS(<value>) — Returns absolute value of a number. — 12-38

ARCTAN(<value>) — Returns the arctangent of a value. — 12-39

BLOCKREAD(<fd>,<array ident>,<block count>|,first block|) — 12-9
Transfers data from a file into an array and returns the count of the number of bytes actually read.

BLOCKWRITE(<fd>,<array ident>,<block count>|,first block|) — 12-10
Transfers data from an array into a file and returns the count of the number of bytes that were actually transferred.

CASE <case selector> OF <list>:|statement|;|<list>:|statement|;...,...| — 5-17
Transfers control to one of several statements labels depending on the variables value.

CHR(<i>) — Returns a character value with the ordinal number y. — 6-8

CLOSE(<fd>|,PURGE|) — Closes and deletes files. — 12-11

CONCAT(<string1>,<string2>|,string3,...|) — 12-2
Concatenates two or more strings.

COPY(<string>,<index>,<size>) — Copies all or part of a string. — 12-3

COS(<value>) — Returns the cosine of value — 12-40

<table>
<tr><th>Reference & Format</th><th>Use</th><th>Page</th></tr>
<tr><td>DATE</td><td>Returns the current date.</td><td>12-53</td></tr>
<tr><td>DELETE(<string>,<index>,<value>)</td><td>Deletes characters from a string.</td><td>12-4</td></tr>
<tr><td>DISPOSE(<ptr>)</td><td>Returns allocated memory to the heap</td><td>12-54</td></tr>
<tr><td>EOF(<fd>)</td><td>Determines whether the end of a file has been reached.</td><td>12-12</td></tr>
<tr><td>EOLN(<fd>)</td><td>Determines whether the end of the line has been reached.</td><td>12-13</td></tr>
<tr><td>EOLNCHR</td><td>Returns an integer value representing a terminator.</td><td>12-55</td></tr>
<tr><td>EXIT</td><td>Results in an orderly exit.</td><td>12-57</td></tr>
<tr><td>EXP(<value>)</td><td>Returns the exponential function.</td><td>12-41</td></tr>
<tr><td>FILLCHAR(<array>,<character>,<length></td><td>Places a character into a packed array a specified number of times.</td><td>12-30</td></tr>
<tr><td>FOR <control var>:=<initial value> DOWNTO <final value> DO <statement></td><td>Executes a simple or compound statement a predetermined number of times.</td><td>5-10</td></tr>
<tr><td>FORWARD:</td><td>Enables a procedure or function to be accessed before it is defined.</td><td>11-2</td></tr>
<tr><td>FUNCTION <ident>|(list:type;...)|:<type>;</td><td>Returns a value.</td><td>11-5</td></tr>
</table>

<table>
<tr><th>Reference & Format</th><th>Use</th><th>Page</th></tr>
<tr><td>GET(<fd>)</td><td>Reads data from a file.</td><td>12-14</td></tr>
<tr><td>GOTO<label></td><td>Unconditionally transfers control from one portion of the program to another.</td><td>5-21</td></tr>
<tr><td>GOTOXY(<x-coord>,<y-coord>)</td><td>Places cursor at specified coordinates.</td><td>12-58</td></tr>
<tr><td>GRAPH(FGCIRCLE,<x-coord>,<y-coord>,<length>)</td><td>Draws a circle or arc with x,y as center and length as the number of pixels to be set.</td><td>17-6</td></tr>
<tr><td>GRAPH(FGCTL,<color group>)</td><td>Selects the color group to be used in high resolution graphics.</td><td>17-8</td></tr>
<tr><td>GRAPH(FGDRAW,<BUFF>)</td><td>Draws a shape on the screen.</td><td>17-10</td></tr>
<tr><td>GRAPH(FGERASE,<BUFF>)</td><td>Erase a shape from the screen by drawing the shape in the background color.</td><td>17-24</td></tr>
<tr><td>GRAPH(FGFILL,<x-coord>,<y-coord>|<,color>|)</td><td>Fills the screen with a specified color until x,y.</td><td>17-26</td></tr>
<tr><td>GRAPH(FGFPOINT,<x-coord>,(y-coord>|<,color>|)</td><td>Returns the value of the point x,y</td><td>17-27</td></tr>
</table>

Reference & Format — Use — Page

GRAPH(FGGET,<x-coord>,<,y-coord>,<BUFF>)

Store the parameters of a rectangle in an array. x,y are the coordinates of the opposite corner from a specified point. — 17-28

GRAPH(FGLINE,<x-coord>,<y-coord>|<,color>|)

Draws a line from the previous point set to x,y in the specified color. — 17-34

GRAPH(FGPAINT,<x-coord>,<y-coord>|<,color>|)

Fills in the pixels with the specified color from x,y to perimeter of the shape on screen. — 17-36

GRAPH(FGPOINT,<x-coord>,<y-coord>|<,color>|)

Sets a point in a specified color. — 17-38

GRAPH(FGPUT,BUFF)

Restore a rectangle to the screen after allow user charges in parameters. — 17-39

GRAPH(FGROT,<degrees>);

Rotates a shape. — 17-41

GRAPH(FGSCALE,<x-dimen>,<y-dimen>)

Multiplies the x-dimension and/or y-dimension. — 17-44

GRAPH(TXFPOINT,<x-coord>,<y-coord>,<value>)

Returns a value of zero if the point x,y is clear otherwise a value of one if point x,y is set. — 16-28

| Reference & Format | Use | Page |
|---|---|---|
| GRAPH(TXPOINT,<x-coord>,<y-coord><,value>) | Sets a point x,y on the screen in low resolution. | 16-26 |
| HALT | Terminates the execution of a PASCAL program. | 12-59 |
| IAND(VALUE,VALUE2) | Returns an integer result from the operation A AND B. | 12-73 |
| IF <cond.expr> THEN <statement> \|ELSE <statement> \| | Executes a specified statement depending on a stated condition. | 5-14 |
| INP(<PORT>) | Function to read a port. | 12-70 |
| INSERT(<source>,<dest>,<index>) | Inserts characters into a string. | 12-5 |
| IOR(A,B) | Returns an integer result from the operation A OR B. | 12-74 |
| IORESULT | Returns the I/O code result of last I/O operations. | 12-15 |
| ISHIFT(A,B) | Returns an integer result from the operation of shifting A. | 12-75 |
| ISWAP(A) | Returns an integer with the low and high byte swaped. | 12-76 |
| IXOR(A,B) | Returns an integer result from the operation A AND B. | 12-77 |

| Reference & Format | Use | Page |
|---|---|---|
| LENGTH(<string>) | Returns the number of characters in a string. | 12-6 |
| LN(<value>) | Returns the natural log of a value. | 12-42 |
| LOG(<value>) | Returns the log (base 10) of a number. | 2-43 |
| MARK(<pointer var>) | Sets a pointer to the top-of-heap of the available free memory. | 12-60 |
| <value1> MOD <value2> | Finds the remainder when two integers are divided. | 12-44 |
| MOVELEFT(<string1>,<string2>,<length>) | Moves a specified number of characters from the left end of one string to the left end of the other. | 12-32 |
| MOVERIGHT(<string1>,<string2>,<length>) | Moves a specific number of characters from one string to another string from the right. | 12-33 |
| NEW(<ptr>) | Allocates space from the heap | 12-62 |
| ODD(<value>) | Determines whether an integer is odd. | 12-46 |
| OPTION | Returns switch options in effect when PASSYS or PASCAL was executed. | 12-63 |
| ORD(<y>) | Returns the ordinal value of the constant y. | 7-2 |

<table>
<tr><th>Reference & Format</th><th>Use</th><th>Page</th></tr>
<tr><td>OUT(<PORT>,<VALUE>)</td><td>Procedure to write to a port.</td><td>12-71</td></tr>
<tr><td>PAGE(<fd>)</td><td>Sends a top-of-form character to a file.</td><td>12-16</td></tr>
<tr><td>POS(<pattern>,<string>)</td><td>Returns the position of the first character of the first occurrence of a pattern in the string.</td><td>12-7</td></tr>
<tr><td>PRED(y)</td><td>Returns the element in a list preceding y.</td><td>7-2</td></tr>
<tr><td>PROCEDURE <ident>|<list:type;...|;</td><td>Manipulates data structures.</td><td>11-3</td></tr>
<tr><td>PWROFTEN(<value>)</td><td>Returns a REAL result of the number 10 raised to the power of the integer parameter supplied.</td><td>12-72</td></tr>
<tr><td>PUT(<fd>)</td><td>Writes a buffer to a file.</td><td>12-17</td></tr>
<tr><td>READ(<fd>|,variable list|)</td><td>Reads data from a file or the keyboard and assigns it to a variable list.</td><td>12-18</td></tr>
<tr><td>READLN |(fd)|<br>READLN(<fd>|,variable list)</td><td>Reads a line of input.</td><td>12-20</td></tr>
<tr><td>RELEASE(<pointer var>)</td><td>Sets the top-of-heap pointer to the memory location of the <pointer var>.</td><td>12-64</td></tr>
</table>

<table>
<tr><th>Reference & Format</th><th>Use</th><th>Page</th></tr>
<tr><td>REPEAT|statement|;statement;...||</td><td>Executes a statement or statement block repeatedly until a deserved condition is met.</td><td>5-8</td></tr>
<tr><td>REWRITE(<fd>,<title>)</td><td>Creates and prepares a file for writing.</td><td>12-22</td></tr>
<tr><td>ROUND(<value>)</td><td>Converts a REAL value into an integer by rounding it to the closest integer.</td><td>12-47</td></tr>
<tr><td>SCAN(<length>),<partial expr>,<array>)</td><td>Returns the number of bytes between a specified starting point in a string and a particular character.</td><td>12-35</td></tr>
<tr><td>SEEK(<fd>, <record number>)</td><td>Allows a file to be read or written starting at a particular record.</td><td>12-23</td></tr>
<tr><td>SIN(<value>)</td><td>Returns the sine of a value.</td><td>12-48</td></tr>
<tr><td>SIZEOF(<identifier>)</td><td>Returns the number of bytes in memory that are assigned to an identifier.</td><td>12-65</td></tr>
<tr><td>SQRT(<value>)</td><td>Returns the square root of a number.</td><td>12-50</td></tr>
<tr><td>SQR(<value>)</td><td>Returns the square of a value.</td><td>12-49</td></tr>
<tr><td>STARTPAR</td><td>Holds the characters that are written after the code - filename - when a program is executed.</td><td>12-66</td></tr>
</table>

<table>
<tr><th>Reference & Format</th><th>Use</th><th>Page</th></tr>
<tr><td>SUCC(y)</td><td>Returns the element in a list succeeding y.</td><td>7-2</td></tr>
<tr><td>SVC(<n>,<parameter block>)</td><td>Executes a supervisor call.</td><td>12-67</td></tr>
<tr><td>TIME</td><td>Returns the time since the system was last booted.</td><td>12-69</td></tr>
<tr><td>TRUNC(<value>)</td><td>Converts a REAL value into an integer by truncating the decimal portion of the number.</td><td>12-51</td></tr>
<tr><td>TYPE <typeid> = (<list>);<br>|typeid = (list);|...,...</td><td>Declares a set of constant values that a variable may assume.</td><td>7-1</td></tr>
<tr><td>VAR <stringid> = STRING [max length];<br>or<br>VAR <ident>|,ident,...|:<pointer id>;</td><td>Declares a string.</td><td></td></tr>
<tr><td>WHILE<cond expr.> DO <statement></td><td>Executes a statement repeatedly until the condition being tested becomes false.</td><td>5-6</td></tr>
<tr><td>WRITE(<fd>|,item list|)</td><td>Outputs variables and strings to the screen.</td><td>12-25</td></tr>
<tr><td>WRITELN|(<fd>)|<br>WRITELN(|<fd>,item list)</td><td>Outputs a line and carriage return.</td><td>12-27</td></tr>
</table>

1

# APPENDIX B
# COMPILE TIME OPTIONS

# SECTION B
## COMPILE TIME OPTIONS

Compile time options are switches that can be set at compile-time from within a program. The format to set an option switch is:

```
(*$<option>)
```

This "dollar sign comment" should appear near the beginning of the program. The following options are available:

| Option | Function |
|---|---|
| D | Numbers program statements. |
| G | Sets the GOTO statement switch. |
| I | Includes an outside source file at compilation. |
| L | Generates a program listing. |
| R | Performs range checking. |

Each switch is described in detail on succeeding pages.

D-Option

Function: Sets the Debugging switch.

Format: (* $<D-switch>*)

Arguments: The D- switch may be:

D+ = statement numbers are generated.

D- = statement numbers are not generated.

Use: The compiler generates statement numbers that are then displayed when a run-time error occurs. This is especially useful in debugging programs.

Note: Extra code is generated so the program will be enlarged.

Example: (* $D+ *)

G-Option

Function: Sets the GOTOOK switch.

Format: (* $<G-switch>*)

Arguments: The G- switch may have two forms:

G+ = Allows GOTO statements.

G- = Enables GOTO statement in a user's program to generate a syntax error.

Use: The switch must be set if a GOTO statement appears in the source.

Note: G- is the switch's default value.

Example: (* $G+ *)

## I-Option

Function: Sets the include switch.

Format: (* $I <fd> *)

Argument: fd is the file descriptor for the source file to be included.

Use: The I-option provides for compiling more than one source file at one time.

Note: The fd should be a PASCAL source file.

Example: (* $I DATA:zestfile *)

## L-Option

Function: Sets the listing switch.

Format: (*$L <fd> *)

Argument: fd is the file descriptor for the output file.

Use: The L-option generates a listing of the source text to the output file.

Note: "PR:" and "CON:" are the most frequently used output files.

Example: (* $L PR:*)

R-Option

Function: Sets the rangecheck switch.

Format: (* $<R-switch>*)

Argument: The R- switch may be the form.
R+ = turns rangechecking on.
R- = turns rangechecking off.

Use: Rangechecking checks if a subscript goes out of range and halts execution on the program. This helps avoid some of the unpredictable errors resulting from subscripts going out of range.

Note: R+ is the option's default value.

Example: (*$R+ *)

# APPENDIX C
# COMPILER ERRORS

## APPENDIX C
## COMPILE-TIME ERROR MESSAGES

| Number | Message |
|---|---|
| 1 | ERROR IN SIMPLE TYPE |
| 2 | IDENTIFIER EXPECTED |
| 3 | 'PROGRAM' EXPECTED |
| 4 | ')' EXPECTED |
| 5 | ':' EXPECTED |
| 6 | ILLEGAL SYMBOL (POSSIBLY MISSING ';' ON LINE ABOVE) |
| 7 | ERROR IN PARAMETER LIST |
| 8 | 'OF' EXPECTED |
| 9 | '(' EXPECTED |
| 10 | ERROR IN TYPE |
| 11 | 'A' EXPECTED |
| 12 | 'A' EXPECTED |
| 13 | 'END' EXPECTED |
| 14 | ';' EXPECTED (POSSIBLY ON LINE ABOVE) |
| 15 | INTEGER EXPECTED |
| 16 | '=' EXPECTED |
| 17 | 'BEGIN' EXPECTED |
| 18 | ERROR IN DECLARATION PART |
| 19 | ERROR IN <FIELD-LIST> |
| 20 | '.' EXPECTED |
| 21 | '*' EXPECTED |
| 22 | 'INTERFACE' EXPECTED |
| 23 | 'IMPLEMENTATION' EXPECTED |
| 24 | 'UNIT' 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 |

| Number | Message |
|---|---|
| 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 | RE-SPECIFIED PARAMS NOT OK FOR A FORWARD DECLARED PROCEDURE |
| 120 | FUNCTION RESULT TYPE MUST BE SCALAR, SUBRANGE POINTER |
| 121 | FILE VALUE PARAMETER NOT ALLOWED |
| 122 | A FORWARD DECLARED FUNCTION'S RESULT TYPE CAN'T BE RE-SPECIFIED |
| 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 |

| Number | Message |
|---|---|
| 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 OS 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/PUNC NOT ALLOWED |
| 165 | MULTIDEFINED LABEL |
| 166 | MULTIDECLARED LABEL |
| 167 | UNDECLARED LABEL |
| 168 | UNDEFINED LABEL |

| Number | Message |
|---|---|
| 169 | ERROR IN BASE SET |
| 170 | VALUE PARAMETER EXPECTED |
| 171 | STANDARD FILE WAS RE-DECLARED |
| 172 | UNDECLARED EXTERNAL FILE |
| 174 | PASCAL FUNCTION OR PROCEDURE EXPECTED |
| 182 | NESTED UNITS NOT ALLOWED |
| 183 | EXTERNAL DECLARATION NOT ALLOWED AT THIS NESTING LEVEL |
| 184 | EXTERNAL DECLARATION NOT ALLOWED IN INTERFACE SECTION |
| 185 | SEGMENT DECLARATION NOT ALLOWED IN UNIT |
| 186 | LABELS NOT ALLOWED IN INTERFACE SECTION |
| 187 | ATTEMPT TO OPEN LIBRARY UNSUCCESSFUL |
| 188 | UNIT NOT DECLARED IN PREVIOUS USES DECLARATION |
| 189 | 'USES' NOT ALLOWED AT THIS NESTING LEVEL |
| 190 | UNIT NOT IN LIBRARY |
| 191 | NO PRIVATE FILES |
| 192 | 'USES' MUST BE IN INTERFACE SECTION |
| 193 | NOT ENOUGH ROOM FOR THIS OPERATION |
| 194 | COMMENT MUST APPEAR AT TOP OF PROGRAM |
| 195 | UNIT NOT IMPORTABLE |
| 201 | ERROR IN REAL NUMBER - DIGIT EXPECTED |
| 202 | STRING CONSTANT MUST NOT EXCEED SOURCE LINE |
| 203 | INTEGER CONSTANT EXCEEDS RANGE |
| 204 | 8 OR 9 IN OCTAL NUMBER |
| 205 | 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 |
| 254 | TOO MANY LONG CONSTANTS IN THIS PROCEDURE |
| 256 | TOO MANY EXTERNAL REFERENCES |
| 257 | TOO MANY EXTERNALS |
| 258 | TOO MANY LOCAL FILES |
| 259 | EXPRESSION TOO COMPLICATED |
| 300 | DIVISION BY ZERO |
| 301 | NO CASE PROVIDED FOR THIS VALUE |
| 302 | INDEX EXPRESSION OUT OF BOUNDS |
| 303 | VALUE TO BE ASSIGNED IS OUT OF BOUNDS |

| Number | Message |
|---|---|
| 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 | CALL NOT ALLOWED IN SEPARATE PROCEDURE |
| 406 | INCLUDE FILE NOT LEGAL |

# APPENDIX D
# RUN TIME ERRORS

# APPENDIX D
# RUN TIME ERRORS

| Number | Message |
|---|---|
| 1 | Format error on p-code file. |
| 2 | Instruction not implemented. |
| 3 | System I/O-error. |
| 4 | End of memory. |
| 5 | Exiting procedure never called. |
| 6 | Integer overflow. |
| 7 | Division by zero. |
| 8 | Floating point error. |
| 9 | String overflow. |
| 10 | Invalid index out of range. |
| 11 | User I/O-error. |

# APPENDIX E
# SUMMARY OF OPERATIONS

## APPENDIX E
## SUMMARY OF OPERATIONS

| Operator | Operation | Type of Operand(s) | Result Type |
|---|---|---|---|
| := | Assignment | Any type except file types | --- |
| **Arithmetic:** | | | |
| + (unary) | Identity | Integer or Real | Same as operand |
| - (unary) | Sign Inversion | Integer or Real | |
| + | Addition | Integer or Real | Integer or Real |
| - | Subtraction | Integer or Real | |
| * | Multiplication | Integer or Real | |
| **Integer:** | | | |
| div | Division | Integer | Integer |
| / | Real division | Integer or real | Real |
| mod | Modules | Integer | Integer |
| **Relational:** | | | |
| = | Equality | Scalar, String | Boolean |
| <> | Inequality | Set or Pointer | Boolean |
| < | Less Than | Scalar or String | Boolean |
| > | Greater Than | Scalar or String | Boolean |
| <= | Less Than or Equal to | Scalar or String | Boolean |
| | -or- | | |
| | Set Inclusion | Set Subset | Boolean |
| >= | Greater Than or Equal to | Scalar or String | Boolean |
| | -or- | | |
| | Set Inclusion | Set | Boolean |
| in | Set Membership | First operand is any scalar, second is its set type | Boolean<br>Boolean<br>Boolean |
| **Logical:** | | | |
| not | Negation | Boolean | Boolean |
| or | Disjunction | Boolean | Boolean |
| and | Conjunction | Boolean | Boolean |
| **Set:** | | | |
| + | Union | Boolean | [] or [A,B,...] |
| - | Difference | Boolean | Boolean |
| * | Intersection | Boolean | Boolean |

# APPENDIX F
# ASCII CHARACTER SET

# APPENDIX F
# ASCII CHARACTER SET

| Dec. | Hex. | Char. | Dec. | Hex. | Char. | Dec. | Hex. | Char. |
|---|---|---|---|---|---|---|---|---|
| 0 | 00 | (NUL) | 43 | 2B | + | 86 | 56 | V |
| 1 | 01 | (SOH) | 44 | 2C | , | 87 | 57 | W |
| 2 | 02 | (STX) | 45 | 2D | - | 88 | 58 | X |
| 3 | 03 | (ETX) | 46 | 2E | . | 89 | 59 | Y |
| 4 | 04 | (EOT) | 47 | 2F | / | 90 | 5A | Z |
| 5 | 05 | (ENQ) | 48 | 30 | 0 | 91 | 5B | [ |
| 6 | 06 | (ACK) | 49 | 31 | 1 | 92 | 5C | \ |
| 7 | 07 | (BEL) | 50 | 32 | 2 | 93 | 5D | ] |
| 8 | 08 | (BS) | 51 | 33 | 3 | 94 | 5E | ^ |
| 9 | 09 | (TAB) | 52 | 34 | 4 | 95 | 5F | _ |
| 10 | 0A | (NL) | 53 | 35 | 5 | 96 | 60 | ` |
| 11 | 0B | (VT) | 54 | 36 | 6 | 97 | 61 | a |
| 12 | 0C | (FF) | 55 | 37 | 7 | 98 | 62 | b |
| 13 | 0D | (CR) | 56 | 38 | 8 | 99 | 63 | c |
| 14 | 0E | (SO) | 57 | 39 | 9 | 100 | 64 | d |
| 15 | 0F | (S1) | 58 | 3A | : | 101 | 65 | e |
| 16 | 10 | (DLE) | 59 | 3B | ; | 102 | 66 | f |
| 17 | 11 | (DC1) | 60 | 3C | < | 103 | 67 | g |
| 18 | 12 | (DC2) | 61 | 3D | = | 104 | 68 | h |
| 19 | 13 | (DC3) | 62 | 3E | > | 105 | 69 | i |
| 20 | 14 | (DC4) | 63 | 3F | ? | 106 | 6A | j |
| 21 | 15 | (NAK) | 64 | 40 | @ | 107 | 6B | k |
| 22 | 16 | (SYN) | 65 | 41 | A | 108 | 6C | l |
| 23 | 17 | (ETB) | 66 | 42 | B | 109 | 6D | m |
| 24 | 18 | (CAN) | 67 | 43 | C | 110 | 6E | n |
| 25 | 19 | (EM) | 68 | 44 | D | 111 | 6F | o |
| 26 | 1A | (SUB) | 69 | 45 | E | 112 | 70 | p |
| 27 | 1B | (ESC) | 70 | 46 | F | 113 | 71 | q |
| 28 | 1C | (FS) | 71 | 47 | G | 114 | 72 | r |
| 29 | 1D | (GS) | 72 | 48 | H | 115 | 73 | s |
| 30 | 1E | (RS) | 73 | 49 | I | 116 | 74 | t |
| 31 | 1F | (US) | 74 | 4A | J | 117 | 75 | u |
| 32 | 20 | (space) | 75 | 4B | K | 118 | 76 | v |
| 33 | 21 | ! | 76 | 4C | L | 119 | 77 | w |
| 34 | 22 | " | 77 | 4D | M | 120 | 78 | x |
| 35 | 23 | # | 78 | 4E | N | 121 | 79 | y |
| 36 | 24 | $ | 79 | 4F | O | 122 | 7A | z |
| 37 | 25 | % | 80 | 50 | P | 123 | 7B | { |
| 38 | 26 | & | 81 | 51 | Q | 124 | 7C | \| |
| 39 | 27 | ' | 82 | 52 | R | 125 | 7D | } |
| 40 | 28 | ( | 83 | 53 | S | 126 | 7E | ~ |
| 41 | 29 | ) | 84 | 54 | T | 127 | 7F | (DEL) |
| 42 | 2A | * | 85 | 55 | U | | | |

# APPENDIX G
# SAMPLE PROGRAMS

# APPENDIX G
# SAMPLE PROGRAMS

This Appendix contains the following sample programs:

1. ISAM Program - ISAMDEMO
2. FGDRAW Sample Programs - DRAWGARPHICS and PUT&GETSHAPE
3. Animation Program - ANIMATESTICK
4. Running Assembly Language Programs under PASCAL
5. Multi-tasking Example.

## G.1 ISAM PROGRAM ISAMDEMO

```
        program isamdemo(input,output);

const
    clearscreen  =  '(:12:)';               (*character for formfeed*)
    bell         =  '(:7:)';
    tensp        =  '          ';           (*ten spaces*)
    twentysp     =  '                    '; (*twenty spaces*)

type
    s80          =  string[80];
    s30          =  string[30];
    s20          =  string[20];

var
    rec, recnew  :  s80;
    name, number :  s30;
    dept         :  s20;
    outfile      :  isamfile;
    finished,errflag : boolean;
    choicea,choiceb  :  char;

PROCEDURE CHECK; forward;

PROCEDURE PRESSIT;                          (*procedure used to pause*)
                                            (*execution of pgm till*)
begin                                       (*<return> is pressed*)
    writeln;
    writeln('Press <return> to continue');
    readln(choicea)
end; (*pressit*)
```

```
PROCEDURE PADREC;                          (*procedure used to construct*)
                                           (*record NEWREC with length of*)
var                                        (*exactly 80 characters*)
    stl, st2  :  s30;
    st3       :  s20;

begin
    stl  :=  concat(twentysp,tensp); (*construct string of 30 spaces*)
    st2  :=  stl;                    (*another string of 30 spaces*)
    st3  :=  twentysp;               (*construct string of 20 spaces*)
    delete(stl,l,length(name));
    stl  :=  concat(name,stl);       (*pad stl with spaces*)
    delete(st2,l,length(number));
    st2  :=  concat(number,st2);     (*pad st2 with spaces*)
    delete(st3,l,length(dept));
    st3  :=  concat(dept,st3);       (*pad st3 with spaces*)
    recnew  :=  concat(stl,st2,st3)  (*construct fixed length record*)
end; (*padrec*)

PROCEDURE CHECK;                           (*procedure used to check for
                                            ISAM errors and return  message*)

var
  x : integer;

begin
  x := ioresult;
  if x = 0 then exit(check);
  if ((x < 120) and (x > 126)) then
  begin
    writeln('ioresult is ',x);
    exit(isamdemo)
  end;
  case x of
       120 : writeln('ISAM ERROR - Key not found',bell);
       121 : writeln('ISAM ERROR - Duplicate key',bell);
       122 : writeln('ISAM ERROR - Illegal key value',bell);
       123 : writeln('ISAM ERROR - Mismatch at check read',bell);
       124 : writeln('ISAM ERROR - Index not found,'bell);
       125 : writeln('ISAM ERROR - Bad data record length',bell);
       126 : writeln('ISAM ERROR - Task : end of memory',bell)
  end;
  errflag := true;
end;
```

```
PROCEDURE RECREAD;  (*procedure used to read in record from file*)

var
    search  :  s30;
    choice  :  s20;
begin
    writeln(clearscreen);
    writeln('Read a record');
    writeln;
    writeln;
    writeln('Choose key : name, number, or dept');
    readln(choice);
    writeln;
    writeln('Do you want to');
    writeln('     L   Read last record');
    writeln('     F   Read first record');
    writeln('     P   Read previous record');
    writeln('     N   Read next record');
    writeln('     S   Search for a record');
    writeln;
    readln(choicea);
    case choicea of
         'L', 'l'  :  isam(outfile,readlast,rec,choice);
         'F', 'f'  :  isam(outfile,readfirst,rec,choice);
         'P', 'p'  :  isam(outfile,readprevious,rec,choice);
         'N', 'n'  :  isam(outfile, readnext,rec,choice);
         'S', 's'  :  begin
                        writeln('String to be searched for');
                        readln(search);
                        isam(outfile,readkey,rec,choice,search)
                      end
check;
if errflag then
begin
  errflag := false;
  pressit;
  exit(recread)
end;

writeln(clearscreen);
```

```
    writeln('The record is');
    writeln;
    writeln;
    writeln(rec);
    pressit
end; (*recread*)

PROCEDURE RECWRITE;  (*procedure used to write a record into a file*)

begin
    writeln(clearscreen);
    writeln('Write a record');
    writeln;
    writeln('Name');
    readln(name);
    writeln;
    writeln('Phone');
    readln(number);
    writeln;
    writeln('dept');
    readln(dept);                        (*now that all the info is in...*)
    padrec;                              (*create the fixed length record*)
    isam(outfile,write,recnew);
    check;
    if errflag then
    begin
      errflag := false;
      pressit;
      exit(recwrite)
    end;
    writeln;
    writeln('Record written!');
    pressit
end; (*recwrite*)
```

```
PROCEDURE RECUPDATE
begin
    writeln(clearscreen);
    writeln('Update a record');
    writeln('First we must read in a record');
    pressit;
    recread;                         (*record must be read in before*)
    writeln;                         (*it can be updated*)
    writeln('New name');
    readln(name);
    writeln;
    writeln('New phone number');
    readln(number);
    writeln;
    writeln('New department');
    readln(dept);                    (*now that all info is in...*)
    padrec;                          (*create the fixed length record*)
    isam(outfile,update,rec,recnew);
    check;
    if errflag then
    begin
      errflag := false;
      pressit;
      exit(recupdate)
    end;
    writeln('Record updated!');
    rec := concat(twentysp,twentysp,twentysp,twentysp);
    recnew :=rec;
    close(outfile);
    reset(outfile,'-:rctest');       (*forces clearing of buffer*)
    pressit
end; (*recupdate*)
```

```
PROCEDURE RECDEL;  (*procedure used to delete record from the file*)

begin
    writeln(clearscreen);
    writeln('Delete a record');
    writeln;
    writeln('First we must read in record');
    pressit;
    recread;                          (*record must be read in before*)
    isam(outfile,delete,rec);         (*it can be deleted*)
    check;
    if errflag then
    begin
      errflag := false;
      pressit;
      exit(recdel)
    end;
    rec := concat(twentysp,twentysp,twentysp,twentysp); (*clear string*)
    close(outfile);
    reset(outfile,'-:rctest');        (*forces clearing of the buffer*)
    writeln;
    writeln('Record was deleted!');
    pressit
end; (*recdel*)

begin (*main program*)
    reset(outfile,'-:rctest');        (*open isam file*)
    finished := false;
    errflag := false;
    repeat
         writeln(clearscreen);
         writeln('ISAM demo');
         writeln;
         writeln;
         writeln('Please choose');
         writeln;
```

```
    writeln('     R   Read record');
    writeln('     W   Write record');
    writeln('     U   Update record');
    writeln('     D   Delete record');
    writeln('     X   Exit program');
    writeln;
    readln(choiceb);
    case choiceb of
        'R', 'r'  :  recread;
        'W', 'w'  :  recwrite;
        'U', 'u'  :  recupdate;
        'D', 'd'  :  recdel;
        'X', 'x'  :  finished  :=  true
        end (*case*)
    until finished
end.
```

## G.2 FGDRAW PROGRAMS - DRAWGRAPHICS AND PUT&GETSHAPE

Interactive program DRAWGRAPHICS draws the shape specified by the user. It consists of two procedures:

| | |
|---|---|
| GENSHAPE | Prompts user for move direction/set information. This data is stored in a buffer. |
| DISPLAYIT | Prompts user for color group, color number and the coordinates of where the shape is to be displayed. The shape stored in a buffer by GENSHAPE is then drawn on the screen. |

Program PUT&GETSHAPE consists of three procedures:

| | |
|---|---|
| INFO | Specifies shape table information. Refer to FGDRAW, Section 17.5, Ex. 2 for shape derivation. |
| WRITE | Writes the shape table data to a disk file. This procedure with little modification can be used in other FGDRAW programs, when required. |
| READ | Reads the disk file containing the shape table information and displays it on the console. This procedure can also be used with little modification in FGDRAW programs, when required. |

Program DRAWGRAPHICS

```
PROGRAM DRAWGRAPHICS;
CONST
  FEED = '(:12:)';
TYPE
  BYTE = 0..255;
  SETBUFF = PACKED RECORD CASE BOOLEAN OF
              TRUE :  (BUF  : ARRAY[0..127] OF INTEGER);
              FALSE : (SIZE : INTEGER;
                       BBUF : PACKED ARRAY[1..254] OF BYTE);
            END;
VAR
  BUFF : SETBUFF;
  IY   : INTEGER;
  RESP : CHAR;

PROCEDURE GENSHAPE
(* READS EACH MOVE AND STORES INTO BUFFER FOR SUBSEQUENT DISPLAY *)
VAR
  X,X1,IX,CB : INTEGER;
  FLAG,MOVE : CHAR;
BEGIN
  IY := 0; FLAG := ' '; MOVE := ' ';
  REPEAT
    FOR IX := 1 TO 2 DO
  BEGIN
    WRITELN(FEED);
    WRITELN('TYPE  u  to  Move up');
    WRITELN('      r  to  Move right');
    WRITELN('      d  to  Move down');
    WRITELN('      l  to  Move left');
    WRITELN('      U  to  Set pixel & move up');
    WRITELN('      R  to  Set pixel & move right');
    WRITELN('      D  to  Set pixel & move down');
    WRITELN('      L  to  Set pixel & move left');
    WRITELN('      0  to  set color 0');
    WRITELN('      1  to  set color 1');
    WRITELN('      2  to  set color 2');
    WRITELN('      3  to  set color 3');
    WRITELN;
    WRITE('Type your move : ');
    READLN(MOVE);
    CASE MOVE OF
      'u'    : X := 0;
      'r'    : X := 1;
      'd'    : X := 2;
      'l'    : X := 3;
      'U'    : X := 4;
      'R'    : X := 5;
      'D'    : X := 6;
      'L'    : X := 7;
      '0'    : X := 12;
      '1'    : X := 13;
      '2'    : X := 14;
      '3'    : X := 15
    END;
```

```
    IF IX = 1 THEN X1 := X
              ELSE CB := X1*16+XX
    END;
    IY := IY + 1;
    BUFF.BBUF[IX] := CB;
    WRITE('Do you want to exit (Y/N)? ');
    READLN(FLAG);
  UNTIL ((IY=254) or (FLAG='Y') OR (FLAG='y'))
END;

PROCEDURE DISPLAYIT; (* Displays shape stored in a buffer *)
VAR
  X,Y,GROUP,NUM : INTEGER;
BEGIN
  BUFF.SIZE := IY;
  WRITELN(FEED);
  WRITE('Enter color group & color number separated by a blank? ');
  READLN(GROUP,NUM);
  WRITE('Where do you want the display to begin (X Y) ? ');
  READLN(X,Y);
  WRITELN(FEED);
  GRAPH(FGCTL,GROUP);
  GRAPH(FGPOINT,X,Y,NUM);
  GRAPH(FGDRAW,BUFF.BUF)
END;

BEGIN
  REPEAT
    WRITELN(FEED);
    GRAPH(FGCTL,2);
    GRAPH(FGPOINT,0,0,0);
    GRAPH(FGFILL,239,239);
    GENSHAPE;
    DISPLAYIT;
    WRITE('Do you want to draw another shape (Y/N) ? ');
    READLN(RESP);
  UNTIL ((RESP='N') OR (RESP='n'))
END.
```

Program PUT&GETSHAPE

```
PROGRAM PUT&GETSHAPE
CONST
  FEED = '(:12:)';
VAR
  BUFF : ARRAY[0..6] OF INTEGER;
  STOSHAPE : FILE OF INTEGER;
  FILENAME : STRING;
  I,L : INTEGER;

PROCEDURE INFO;
BEGIN
  (* Fill out shape table information *)
  BUFF[0] := 12;                  (* Length in bytes *)
  BUFF[1] := 65*256+4;
  BUFF[2] := 65*256+85;
  BUFF[3] := 82*256+85;
  BUFF[4] := 99*256+102;
  BUFF[5] := 67*256+118;
  FOR I := 0 TO 6 DO              (* Display on console *)
    WRITELN(BUFF[I]);
  WRITELN
END;

PROCEDURE READ;
BEGIN
  RESET(STOSHAPE,FILENAME);       (* Open file *)
  I := 0;
  GET(STOSHAPE);
  WHILE NOT EOF(STOSHAPE) DO      (* Read file & put it into BUFF *)
  BEGIN
    BUFF[I] := STOSHAPE ;
    I := I + 1;
    GET(STOSHAPE)
  END;
  I := I-1; L := I;               (* Manipulate L for displaying *)
  FOR I := 0 TO L DO
    WRITELN(BUFF[I])
END;
```

```
PROCEDURE WRITE;
BEGIN
  REWRITE(STOSHAPE,FILENAME);        (* Create file *)
  IF IORESULT <> 0 THEN
  BEGIN
    RESET(STOSHAPE,FILENAME);        (* Open file *)
    CLOSE(STOSHAPE,PURGE);           (* Close file & delete it *)
    REWRITE(STOSHAPE,FILENAME)       (* Create file *)
  END;
  IF (BUFF[0] MOD 2 = 0) THEN
    L := BUFF[0] DIV 2 ELSE          (* If length is even bytes *)
    L := BUFF[0] DIV 2 + 1;          (* If length is odd bytes *)
  FOR I := 0 TO L DO
  BEGIN
    STOSHAPE  := BUFF[I];
    PUT(STOSHAPE)                    (* Write to file *)
  END;
  CLOSE(STOSHAPE)                    (* Close file *)
END;
BEGIN
  WRITELN(FEED);
  INFO;
  FILENAME := '-:SHAPETABLE';
  WRITE;                             (* Write to file *)
  L := 0;
  FOR I := 0 TO 6 DO                 (* Put 0s in BUFF to check for
                                        reading of file *)

    BUFF[I] := 0;
  READ                               (* Read file *)
END.
```

## G.3 ANIMATION PROGRAM - ANIMATESTICK

This program draws a blue bar on the left side of the screen. It then moves the bar across the screen from X=1 to X=230 position.

```
PROGRAM ANIMATESTICK;
CONST
  FEED = '(:12:)';
VAR
  X,Y,C : INTEGER;

PROCEDURE DRAW;
BEGIN
  GRAPH(FGPOINT,X,Y,C);
  GRAPH(FGLINE,X,Y+100);
  X := X + 2
END;

PROCEDURE ERASE;
BEGIN
  GRAPH(FGPOINT,X-4,Y,0);
  GRAPH(FGLINE,X-4,Y+100);
END;

BEGIN
  WRITELN(FEED);                      (* Clear low res screen *)
  GRAPH(FGCTL,0);                     (* Clear                *)
  GRAPH(FGPOINT,0,0,0);               (*       hi-res         *)
  GRAPH(FGFILL,239,239);              (*              screen *)
  GRAPH(FGCTL,109);                   (* Select color group *)
  C := 1; X := 1; Y := 1;             (* Initialize values *)
  DRAW;                               (* Draw stick *)
  C := 2;
  REPEAT
    DRAW;
    IF C=1 THEN GRAPH(FGCTL,108)      (* Switch color group *)
    ELSE GRAPH(FGCTL,109);
    ERASE;
    IF C=1 THEN C := 2 ELSE C := 1;  (* C determines color group *)
  UNTIL (X >= 230)
END.
```

## G.4 RUNNING ASSEMBLY LANGUAGE PROGRAMS UNDER PASCAL

SVC6 may be used to run Assembly Language Programs or task files. The SVC6 function requires that a header be initialized prior to running the command. Refer to the parameter block description below for the header locations.

Executing Program SWITCH (see following page) is an example of multi-tasking since the Pascal program and the Assembly language program will run concurrently. Refer to the 8800 Series Monroe Operating System Programmer's Reference Manual for more information on SVC commands.

### Parameter Block

The parameter block for SVC6 is shown below for your convenience.

<table>
<tr><td>(0) SO.FC<br>Function code</td><td>(1) SO.RS<br>Return status</td><td>(2) S6.PRIO<br>Priority</td><td>(3) S6.OPT<br>Option</td></tr>
<tr><td colspan="2">(4) S6.TID<br>Name pointer or task number</td><td colspan="2">(6) S6.PAR<br>Parameter</td></tr>
<tr><td colspan="2">(8) S6.SAD<br>Address</td><td colspan="2">(10) S6.FD<br>File descriptor</td></tr>
<tr><td colspan="2">(12) S6.SIZE<br>Additional size</td><td colspan="2"></td></tr>
</table>

The parameter block for SVC6 has the following structure:

| | Offset | Byte | Type | Mnemonic | Name |
|---|---|---|---|---|---|
| 1) | 0 | 1 | Byte | SO.FC | Function Code |
| 2) | 1 | 1 | Byte | SO.RS | Return Status |
| 3) | 2 | 1 | Byte | SO.PRIO | Priority |
| 4) | 3 | 1 | Byte | S6.OPT | Option |
| 5) | 4 | 2 | Integer | S6.TID | Name Pointer or task number |
| 6) | 6 | 2 | Byte | S6.PAR | Parameter |
| 7) | 8 | 2 | Address | S6.SAD | Address |
| 8) | 10 | 2 | Address | S6.FD | File descriptor |
| 9) | 12 | 2 | Integer | S6.SIZE | Additional size |
| | Total | 14 | | | |

PROGRAM SWITCH

```
PROGRAM SWITCH      (* Running assembly language program under Pascal *)
TYPE
  BYTE = 0..255;
  SVCB = PACKED RECORD
           FC  : BYTE;          (* Function code *)
           RS  : BYTE;          (* Return status *)
           PRI : BYTE;          (* Priority *)
           OPT : BYTE;          (* Option *)
           TID : INTEGER;       (* Pointer to task-id *)
           PAR : INTEGER;       (* Pointer to starting parameter *)
           SAD : INTEGER;       (* Starting address *)
           FD  : INTEGER;       (* Pointer to file descriptor *)
           SIZE : INTEGER;      (* Additional size *)
         END;

VAR
  SW : PACKED ARRAY[0..31] OF 0..1;
       (* Above variable holds start switches for the program at start
          up.  SW[0] := 1 is equal to start switch A and so on... SVC6
          block must follow *)
  SVC6 : SVCB;
  TIDSTR : STRING[4];           (* Hold task-id *)
  PARSTR : STRING;              (* Hold starting parameters *)
  FDSTR : STRING;               (* Hold name of file *)

BEGIN
  FDSTR := '-   COPYLIB                ';
  TIDSTR := 'GO   ';
    (* Task id - four characters *)
  PARSTR := '(:10:)(:0:)-:TSKRUN/A(:0:)(:0:)';
    (* This string holds the start parameters for the task.  First
       two bytes holding the length of string (low,high order).  The
       parameter string follows, string is terminated by two zeros
       to make an even number of bytes for string.  Even number of
       bytes is required to please OS. *)
  SW[3] := 1;

    (* Fill out SVC block. *)
  WITH SVC6 DO
  BEGIN
    FC  := 3;                   (* Load & start *)
    PRI := 0;                   (* Set default value of priority *)
    OPT := 0;                   (* Set default value of option *)
    TID := ADDRESS(TIDSTR)+1;   (* +1 to step length byte of string *)
    FD  := ADDRESS(FDSTR)+1;
    PAR := ADDRESS(PARSTR)+1;   (* Omit if no parameters *)
    SAD := 0;                   (* OS will figure out address *)
    RS  := 0;                   (* Return status not used *)
    SIZE := 0
  END;
```

```
  IF NOT SVC(6,SVC6) THEN WRITELN('START FAILED');
  SVC6.FC := 32+8;
    (* 32+8 is wait for task termination. *)
  IF NOT SVC(6,SVC6) THEN WRITELN('WAIT DOES NOT WORK');
    (* Execution of above statement will put Pascal program into wait
       statement till termination of running task (COPYLIB in this
       case) *)
  WRITELN;
  WRITELN('SUCCESS!')
END.
```

Execution Commands

Compilation

```
-PASSYS ,-:SWITCH¶
00.00.00 MS.8PASCAL 3.02
.
.
.
00.00.00 End of task 0
```

Execution - Sample Run

```
-PASCAL :SWITCH¶
00.00.00 MS.8 Pascal 3.02

00.00.00 GO  COPYLIB  R1-04.
TSKRUN                Asc        Deleted.
00.00.00 GO  End of task 0
SUCCESS!

00.00.00 End of task 0
-
```

## G.5 MULTI-TASKING EXAMPLE

In order to run several tasks concurrently, the CSS-mode command file is used to direct the task execution.

In this comprehensive example which follows, two PASCAL programs, TASKA and TASKB, are converted to type TSKPAS files with the CSS-command file PASTSK. They are subsequently started concurrently with CSS-command file TSKRUN.

### Program file: TASKA

Program TASKA contains the following code:

```
PROGRAM TASKA (OUTPUT);
VAR
OUTPUT:TEXT;
IX:INTEGER;
BEGIN
RESET(OUTPUT,'PR:');
FOR IX:=1 TO 50 DO
WRITELN(OUTPUT,'THIS IS LINE ',IX,' OF 50 LINES TO BE PRINTED');
END.
```

When executed this program prints 50 lines of text on the printer.

### Program file: TASKB

Program TASKB contains the following code:

```
PROGRAM TASKB;
VAR
IX:INTEGER;
BEGIN
FOR IX:=1 TO 500 DO
WRITLEN('TASKB WILL DISPLAY ON THE CONSOLE ',IX,' OF 500 TIMES');
END.
```

When executed this program displays 500 lines of text on the console.

CSS file: PASTSK

The CSS-file PASTSK, shown below, will convert an ASCII PASCAL source file to a task file.

This CSS-file was documented in detail in Section 13 under "Illustrated Examples".

```
$LOGG
$$DI;$$DI Create Pascal Task - CSS 1.00;$$IF NOP(@1);$$FI CON:
Parameter error!
Enter: source-fd, <task-fd>, <task stack-size>,<list-fd>
Default: task-fd=source-fd, task stack-size=2000
$$ENDF;$E;$$ENDI
$$IF F(RLDR/T),F(PASOBJ/T),F(PASRTL/0):$$GO 10;$$ENDI;$$FI CON:
This CSS requires the following files on the system volume:
  RLDR   - Task establisher.
  PASOBJ - Pascal object code generator.
  PASRTL - Pascal run time library.
$$ENDF;$E;$$10:IF NOF(@1/A);$$DI Assign error on @1!;$E;$$ENDI;$T
$$DI Compilation started...;<Q>-:PASCOMP,@1,CSS%B,@4
$$IF E;$$DI Compilation error on file @1!;$E;$$ENDI
$$LO PASOBJ,PO;$$PRI PO,200;$$ST PO CSS%B;$$WA PO;
$$DE CSS%B/B
$$BU CSS%L
$$FI CSS%C
REMOTE
CHECK
OPTION NOSTACK
STACK @2
INC CSS%B
LIB PASRTL
TASK @1
END
$$ENDF;
$$LO RLDR,,10000;$$PRI RLDR,200;$$ST RLDR CMD=CSS%C;$$WA RLDR
$$DE CSS%B/O;$$DE CSS%C/A;$$DI;$$DI Task @1 created!;$P
$$ENDB
/CSS%L,@2,@3;$$DI NO ERRORS
$$DI PARA1 WAS @1;
$$DI PARA2 WAS @2;
$$DI PARA3 WAS @3;
$$DI PARA4 WAS @4;
$E
```

CSS-FILE: TSKRUN

A CSS-command file can be built via the text Editor EDIT to load and start task files TASKA and TASKB and assign equal priorities to each task. This file is shown below.

```
$LOGG
$$DI;$$DI RUN MULTI TASKING
$$LO TSKA;$$PRI  TSKA,1;$$ST TSKA
$$LO TSKB;$$PRI  TSKB,1;$$ST TSKB
$$WA TSKA;$$DI;$$DI TASKA IS FINISHED
$$WA TSKB;$$DI:$$DI TASKB IS FINISHED
$E
```

Command File: POP

A command file can be created via the Text Editor EDIT which contains all the necessary commands in the proper sequence to control the multi-tasking activity. This file, named POP, looks like this:

```
PASC:DELETE LORI:TSKA/TP,LORI:TSKB/TP
PASSYS/PASTSK,TASKA,TSKA,8024,PR;
PASSYS/PASTSK,TASKB,TSKB,8024,PR;
SLICE 100
PASSYS/TSKRUN
```

Execution

To begin execution of this multi-tasking example type.

```
!POP
```

TASKA will print a line of text on the printer for approximately 20 lines of text displayed on the console by TASKB.

Pausing and Canceling Tasks

It is also possible to pause, continue and cancel tasks. By utilizing the TA,F command at system level, the current task and their respective states are displayed.

The following console record shows a pause, "TA,F" and cancel TSKA.

```
-PASSYS/tskrun¶
00.00.00 MS.8 Pascal 3.02

00.00.00 $$DI;$$DI RUN MULTI TASKING
RUN MULTI TASKING

00.00.00 $$LO TSKA;$$PRI TSKA,1;$$ST TSKA
00.00.00 $$LO TSKB;$$PRI TSKB,1;$$ST TSKB
<Pause task via a CTRL-A sequence>
-PA¶
-¶
00.00.00 Paused
<Pause task via a CTRL-A sequence>
-PA¶
-¶
00.00.00 Paused
-PA tska¶
-CA tska¶
-¶
00.00.00 TSKA End of task 255
<Pause task via CTRL-A sequence>
-CA TSKB¶
-¶
00.00.00 TSKB End of task 255
-CA¶
-¶
-¶
00.00.00 End of task 255
```

# APPENDIX L
# RLDR - RELOCATABLE LOADER

1

# APPENDIX L
# RLDR - RELOCATABLE LOADER

## L.1 INTRODUCTION

The Relocatable Loader (or Task Establisher, as it is also known) is a utility program that builds an executable program (i.e., task) from an object program or set of object programs in an object library. The resultant program generated by RLDR is put on disk as a file or type T-"taskasm."

A task may be either relocatable or absolute. Relocatable tasks can be located anywhere in memory. When they are loaded into memory for execution all necessary addresses are modified for the particular memory they will be occupying. An absolute task is a memory-image executable program. Operating system MS8 (Task OS) itself is an example of an absolute task.

RLDR is directed by a command stream which may either be entered interactively or stored in an ASCII file. In the simplest case, no command stream is necessary. As a by-product of the task generation, RLDR can produce a listing of the memory map and global symbols.

The loader works in two passes. In the first pass, acting on commands from the operator or an ASCII command file, it builds on disk a temporary file. This file contains object programs as they are introduced, a symbol table in memory containing the names of global symbols, and as they become defined, their values. When the END command is encountered, RLDR enters the second pass. During this pass the temporary file is read and a task file is built utilizing the values of the symbols as now defined in the system table. At the end of the second pass a list of all global symbols is printed.

Before using RLDR, the PASCAL program must be compiled and the object file must be created via PASOBJ.

## L.2 RLDR INVOCATION

RLDR is invoked by the command:

RLDR,|switches|,memory |CMD=fd|

switches — Optional. The following switches are available:

R - Additional code for range checking will be generated.

O - Additional code for I/O checking will be generated.

Valid entries are F, O, or RO; if one of these is specified a comma must precede it.

memory — Required. Additional memory for symbol table. Specify 20000 or greater.

fd — Optional. Name of an ASCII file containing a command stream. RLDR reads the commands from that file. If the file contains no END command, RLDR will revert to (interactive) command mode after processing all the commands in the file.

## L.3 COMMANDS

Once RLDR is invoked, the following set of commands can be entered. Other commands are available but generally are not required for PASCAL. In the command descriptions the following generic terms are used:

<fd> — File descriptor or (for INCLUDE and LIBRARY commands) module descriptor. (See Section 1.3.)

<exp> Any expression. An expression may be a symbol; or a numeric constant in octal, decimal, or hexadecimal format; or a character constant; or symbols and constants connected by operators:

| | |
|---|---|
| + | plus |
| - | minus |
| / | division |
| * | multiplication |
| & | AND |
| ! | OR |
| ? | XOR |
| # | shift |

Only the operators '+' and '-' can be used with relocatable symbols.

The value of the PLC selected by the PLCNR command can be referenced by the symbol '*' which is always relocatable.

Expressions are evaluated from left to right as 16-bit values.

Command Function

ABort

Abort RLDR and delete the task file.

ABSolute

Generate an absolute task. The default is relocatable.

CHAin fd

Process <fd> for further commands.

| Command | Function |
|---|---|

ENd [ exp ]

Terminate the command phase and initiate the second pass. If <exp> is specified, then that defines the starting address of the task.

INClude[,opt] fd [ ,fd ... ]

Insert an object file or set of object files into the task. Fd may specify either an object library or a "module":

fd.name

where"fd" is the file name of an object library and "name" is the name of an object program within that library.

If fd is an object library then all of its modules will be included, subject to the search rules of the <opt> parameter. If fd is a module name, then only that module will be included.

opt:

- A: Do not load the module, but use the definitions of any absolute symbols that satisfy unresolved entries in the symbol table.
- E: Search through the file until end-of-file is found.
- R: (default) The file is searched from its current position until the end of file; it is then rewound and searching resumed. This continues until no more references can be resolved by the file.
- W: Same as R except the file is rewound before searching starts.

| Command | Function |
|---|---|
| LIBrary PASRTL | Required; collects the modules from the PASCAL Routine Library. |
| LISt [ opt ]<br>NOList [ opt ] | Specify symbols to be included (LIST) or excluded (NOLIST) in the listing file:<br>1. If <opt> is not specified, all symbols.<br>2. If <opt> is ABSolute, all absolute symbols.<br>3. If <opt> is UNUSed, all unreferenced symbols. |
| LOg fd | Produce a log of the commands and error messages to the file <fd>. |
| MAXLu exp | Set the maximum number of logical units the output task can use. The default is 3. |
| MAXNode exp | Sets the maximum number of nodes generated for the output task. The default is 8. |
| OBject fd | Rename the temporary file as <fd> and save it. This is useful for creating a new object library from others by using the INCLUDE and LIBRARY commands. |

| Command | Function |
|---|---|

OPtion opt [ ,opt ... ]

Set the task load options:

| | |
|---|---|
| RESident | Memory resident |
| NONAbortable | Not abortable from other tasks |
| DEFAssign | Default logical unit assignment |
| NOStackcheck | Disables stack limit checks during SVC |
| ERMsg | ERror messages generated by MS8. |

ORG exp

Set the value of the current PLC (selected by the previous PLCNR command) to the value of <exp>.

PAuse

Set RLDR in the pause state.

PRINt fd

Specify the file where the listing is to be written. This command overrides any <List> file specified in the RLDR invocation.

PRIOrity exp

Set the default priority for a task to <exp>. The value of <exp> must be in the range 9-250. The default is 128.

RADix exp

Select the number base for the listing file. The value of <exp> must be 8 or 16. This command overrides a <Radix> specified in the RLDR invocation.

RELocatable

Generate a relocatable task. This is the default case.

| Command | Function |
|---|---|
| REMote | Abort RLDR if an error is detected. |
| STACKlimit exp | Set the task stack size to <exp>. The default value is 256. |
| TAsk fd | Specify the output task file name. This overrides <task> specified in the RLDR invocation. |

## L.4 MESSAGES

RLDR may display the following messages on the console:

ESTAB MS8 Rx.yz<date>
: Sign-on message: revision level x, update level yz, date of last update.

COMMAND ERROR
: Unrecognized command.

PARAMETER ERROR
: Parameter missing or invalid expression.

FILE NAME ERROR
: Invalid fd.

FILE NOT FOUND
: Undefined fd.

MODULE <name> NOT FOUND
: Module not present in the object library file.

| Command | Function |
|---|---|
| END OF TASK <s> | |
| | Termination message. <s> may be: |
| | 0: No errors |
| | 1: Multiply defined or undefined symbols |
| | 2: Aborted |

## L.5 ILLUSTRATED EXAMPLE

Create the Task file DEMOISAM from PASCAL source program ISAMDEMO in Appendix G.

| | |
|---|---|
| PASSYS ,ISAMDEMO | Compile ISAMDEMO |
| PASOBJ ISAMDEMO,DEMOOBJ | Place object code in DEMOOBJ |
| RLDR,,20000 | Invoke RLDR |
| OPTION NOSTACK | Do not perform stack check |
| INC DEMOOBJ | Include object file DEMOOBJ |
| LIB PASRTL | Collects modules from PASCAL Run Time library. |
| TASK DEMOISAM | Links DEMOOBJ into DEMOISAM |
| END | Terminate RLDR |

For additional examples of how to use RLDR refer to Section 13 in the PASOBJ System Program description.

# GLOSSARY OF TERMS

## GLOSSARY OF TERMS

| | |
|---|---|
| Actual parameters | A call to a function or procedure can pass actual parameter. These must be the same in number, sequence, and type as the formal parameters. |
| Array | An ordered collection of values that are all referenced by a single variable name. |
| Array declaration | Sets aside memory space for an array. |
| Array subscript | Designates a particular element of an array. It can be any arithmetic expression that has an integer value. |
| ASCII character set | The American Standard Code for Information Interchange character set. It consists of 128 representations. See Appendix F for more information. |
| Base type | A structured variable's component's type. |
| CHAR | Type CHAR variables have characters as their values. These values can be assigned, compared, read and written. |
| Character | A letter, digit or special character (+, *, \|, etc.). |
| Compiler | A system program that interprets a program in a higher level language, such as PASCAL, into a program in machine code so that the computer may execute it. |

| | |
|---|---|
| Compound statement | A series of simple statements that are preceded by the reserved word BEGIN and followed by an END. |
| Constant definition | Establishes named constants for use in a program. Constants are defined immediately after the program headings. |
| CSS-file (command string supervisor) | A file of commands that are executed by system programs or the PASSYS interpreted. |
| Dynamically allocated variables | Variables whose size and number of components are changed as needed within a program. |
| External files | Files that contain source code that must be linked with other programs before execution. |
| Enumerated type | A type whose values are given by listing their names. |
| Formal parameters | The dummy variables in a function or procedure heading that receive values to be passed during execution. |
| Functions | Subroutines that return a value. |
| Global variables | Variables that are declared in an outer program and are accessible to nested subprocedures. |
| Heading | The first line in a function, procedure or program that contains the routine's name and all applicable parameters. |

| | |
|---|---|
| Identifier | The name used to refer to a variable or constant. It must begin with a letter. |
| Integer constant | An integer (whole number), for example, 200 and 2575. |
| Leading loop decision | A loop with the terminating condition tested for at the top of the loop. |
| Linked list | A linked sequence of total items. |
| Literal | A sequence of characters enclosed in quotes. |
| Local variables | Variables whose values are only accessible within part of a program, i.e., in a subprogram. |
| Logical operators | The value a logical operator returns is either true or false. |
| Machine code | Code that is directly executable by the computer. |
| Maxint | The largest possible INTEGER in PASCAL. |
| Operator precedence | The order in which operators are executed in an expression. |
| P-code (psuedo code) | The code that is outputted from the compiler. |
| Pointer | Each pointer in PASCAL can point to an item of only one type. For example, pointer P can locate only values of type T. |

| | |
|---|---|
| Port | An I/O register that interfaces between the processor and its paripheral services. |
| Procedures | Subroutines that perform tasks and operate on data. They do not return values. |
| Program block | All parts of a program except the heading. |
| Queue | A specialized type of list that allows total items to be added at one end and removed from the other. |
| Range checking | All subscripts are checked at execution to insure that they do not go outside of a specified range of values. |
| Real constant | A number such as 2.414141. |
| Record | A variable with multiple components, each of which may have a different base type. |
| Recursive procedure | A procedure that calls itself. New formal parameters and local variables are allocated each time the procedure is called. |
| Relational operators | Operators that compare values to determine their relative magnitudes. |
| Reserved words | Words that have predefined meanings in PASCAL. |

| | |
|---|---|
| Reset file | Prepare a file for reading by a program. |
| Rewrite file | Prepare a file for writing by a program. |
| Scalar type | A variable type that represents a single value rather than a series of values. For example: CHAR is scalar; STRING Is not. |
| Segmented files | A segmented file is left outside of memory until it is called. Segmented files must be linked together using PASLINK. |
| Sequential files | The elements in sequential files are accessed serially without special file markings. Compare to TEXT files. |
| Set | A collection of values all of the same type. |
| Simple statement | A single PASCAL statement, i.e., not a compound statement. |
| Stack | A list that is restricted to having entries added (pushed) or removed (popped) from the beginning. |
| Standard variable types | PASCAL types that are redefined. There are four: INTEGER, REAL, CHAR, BOOLEAN. |
| Statement block | The group of statements in a program following all declarations. It is preceded by the reserved word BEGIN and followed by END. |

| | |
|---|---|
| String | An array of characters that contains a length value in the first byte. |
| Structured types | Variable types with standardized structures that contain more than a single element. |
| Subrange type | A variable type that can only be assigned a limited range of values. The values are defined in another variable type. |
| Text files | Files of CHAR that contain special characters that mark the end of a line. |
| Top-of-heap | An integer type pointer which is not by MARK and refers to the pointer address at the start of the dynamic data structure called the Heap. |
| Type | Each variable has a specific type whose type is determined by its declaration. |
| Trailing loop decision | A loop with the termination decision made of the end of a loop. |
| Value parameters | Those formal parameters found in a procedure or function heading that are not preceded by the keyword VAR which are passed to the procedure as values. |
| Variable parameters | Those formal parameters in a procedure heading that are preceded by the keyword VAR. Functions may not have variable parameters. |

| | |
|---|---|
| Variable declaration | Establishes variables for use in a program. |
| Variant record | A record that can have a varying structure. The structure is changed within a program. |

1

# INDEX

INDEX

10-0000002-2

# MONROE TEXT EDITOR
# PROGRAMMER'S REFERENCE MANUAL

July 1981

MONROE SYSTEMS FOR BUSINESS
The American Rd.
Morris Plains, N.J. 07950



2105-S

The material contained herein is supplied without representation or warranty of any kind by Monroe Systems For Business. Monroe assumes no responsibility relative for the use of this material and shall have no liability, consequential or otherwise arising from the use of this material or any part thereof. Further, Monroe reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation to notify any person of such revision or changes.

## PURPOSE OF THIS DOCUMENT

This document is a Programmer's Reference Manual. It is to be used by experienced programmers as a reference tool. It is not intended for use as a learning aid by non-programmers.

RECORD OF CHANGES

| Change No. | Date | Pages Affected | Description of Changes |
|---|---|---|---|
| -1 | 7/81 | All | Reviewer's Changes |
| -2 | 7/81 | All | Preliminary Edition |

TABLE OF CONTENTS

TABLE OF CONTENTS (CONT.)

TABLE OF CONTENTS (CONT.)

# SECTION 1
# INTRODUCTION

# SECTION 1
# INTRODUCTION

## 1.1 OVERVIEW

This manual describes the utility program EDIT, a general purpose, text editing program that modifies and/or creates source programs and other ASCII text material (e.g., data, documentation) interactively.

Text is read sequentially from the input unit (terminal) into an area of memory called the edit-buffer. Both the modification and creation of text are performed within the edit-buffer; the former revising the text of a file or segment of a file just read into the edit-buffer, and the latter using the edit-buffer as a work area for the generation of a new text. When editing is complete, the contents of the edit-buffer may be written onto the output unit (printer or file).

EDIT is edit-buffer oriented within the file, line oriented within the edit buffer, and character oriented within a line.

EDIT makes use of any available memory for the edit-buffer.

## 1.2 USING THIS MANUAL

This manual deals with the function and capabilities of the editor and provides the means by which those capabilities can be exploited. Each command is reviewed and examples are provided to facilitate its use. It is important to note, however, that this document is a PROGRAMMER'S REFERENCE MANUAL and not a tutorial. Thus, it is intended to be used as a reference device rather than as an educational tool.

## 1.3 TEXT SYMBOLS AND CONVENTIONS

Throughout this manual specific documentation conventions are used to describe formats for writing EDIT commands and statements. The following conventions are in effect:

| Symbol | Description and Use |
|---|---|
| 1. CAPITAL LETTERS | Capital letters are used for all keywords, commands and statements that are to be explicitly typed.<br>Example: RE<br>OR |
| 2. Lower Case | Lower case letters specify variables which are to be supplied by the user according to the rules explained in this text.<br>Example: line number |
| 3. < > | Angle brackets enclose fields that are required for valid syntax. They are never to be typed.<br>Example: SV <string> |
| 4. , - | Commas and dashes are separators. All must be typed as shown.<br>Example: PR 1-4<br>BT 11, 17, 35 |
| 5. ¶ | The symbol "¶" indicates the depression of the RETURN key.<br>Example: EN¶<br>CV TST1, TST2¶ |
| 6. [] | Vertical lines enclose optional elements of a statement.<br>Example: PR [line nos.] |
| 7. CTRL-H | Control character. Depress and hold CTRL key while striking another key (represented by H).<br>Example: CTRL-C<br>CTRL-H |

## 1.4 FILE-VOLUME-DEVICE-NAMING CONVENTIONS

The file, volume, and device naming conventions that are used throughout this manual are defined as follows.

A) A file is a program or a collection of data stored on a disk-type storage medium. Files stay in the system permanently unless they are explicitly removed.

B) A volume name is the name given by the user to a diskette (e.g., MONT:,PASC:,FIX:). The system volume is the volume from which the operating system is booted.

C) A device name is name given to a physical device (e.g., CON for the console, PR for the printer, FPY0 for disk 0, FPY1 for disk 1). These names cannot be changed by the user.

D) The file/device descriptor, referred to in this manual by <fd>, may refer to any of the above (A, B or C) depending upon the content of the utility command being discussed.

E) File/device descriptors can be composed of four fields: vol, filename, directory, and type, where vol can be either a volume or device name.

F) The format can be expressed in four ways:

1. <vol:>
2. [vol:]<filename>[/type]
3. [vol:]<directory>
4. [vol:]<directory.filename>[/type]

where:

vol — Name of the volume on which the file resides if the file descriptor refers to a file, or the name of a device if the file descriptor refers to a device. It may be from one to four characters. The first character must be alphabetic and the remaining alphanumeric. If the volume is not specified, the default volume is the SYSTEM volume.

| | |
|---|---|
| filename | Name of the file. It may be from one to twelve characters, the first alphabetic and the remaining alphanumeric. |
| directory | Name of the user's directory file. It may be from one to twelve alphanumeric characters. If not specified, the directory defaults to the master directory. |
| type | Type of file, i.e., ASCII, Binary, etc. |

Example: Examples of legal file/device descriptors are:

| | |
|---|---|
| EDIT MAIN | Starts the program EDIT; the file MAIN from the Master Directory is ready to be read into the eidt-buffer. |
| EDIT INFO:MAIN | Starts the program EDIT; the file MAIN from disk "INFO" is ready to be read into the edit-buffer. |
| EDIT INFO:TEST.MAIN | Starts the program EDIT; the file MAIN from user directory "TEST" on disc "INFO" is ready to be read into the edit-buffer. |

## 1.5 KINDS OF FILES

With each file there is a type specification that describes, for the system and the user, what kind of data is in the file. These appear next to the filename for your files in your Master File Directory. Table 1-1 lists these specifications and their meanings. The type of the file is normally implied by the program, and does not need to be specified. If the file type is not implied by the program, it must be specified!

Table 1-1. Type Specifications

| SYMBOL | Description and Use |
|---|---|
| Asm | ASSEMBLER source code. |
| Bas | BASIC source code, or data produced by BASIC. |
| Und | Undefined data, which verifies to any other type. |
| Asc | ASCII data readable without any special handling. |
| Lst | List file, ASCII data together with position information. |
| Obj | Object code, readable by the task Establisher. Cannot be loaded and executed. |
| Bin | Binary data, which is unspecified. |
| Tsk | Task file, either relocatable or absolute. Can be loaded and executed. |
| Ism | ISAM index file. |
| Pas | PASCAL source code or data produced by PASCAL. |
| Efd | Element File Directory. |
| Mfd | Master File Directory. |

## 1.6 ORGANIZATION OF THIS MANUAL

This manual is organized into four sections and two appendices.

Section 2 demonstrates the statements needed to initiate the editor and addresses disc file procedures, modes of work and editor prompting.

Section 3 gives the reader the information necessary to understand and work with the editor. It provides examples which illustrate text creation and modification, text identification, and how to save text and retrieve existing text on a disk file.

Section 4 describes each of the EDIT commands and statements available to the user. For every command or statement, the following information is included:

| | | |
|---|---|---|
| 1. | Function | -Summarizes purpose of the command. |
| 2. | Mode | -Specifies which mode applies - Command or Inline. |
| 3. | Format | -Shows the command syntax. |
| 4. | Arguments | -Defines the format variables. |
| 5. | Use | -Describes in detail how the command is used including restrictions and exceptions. |
| 6. | Example | -Lists program examples illustrating the various uses of the command. |

Appendix A contains a command summary which includes the command mnemonic, its format, and function.

Appendix B contains a list of error messages with comments.

## 1.7 ABBREVIATIONS

The following abbreviations are used in this manual:

| | |
|---|---|
| fd | -File Descriptor |
| line no. | -Line Number |
| PR | -Printer |
| string var | -String Variable |
| vol | -Volume Name |

## 1.8 RELATED MANUALS

This document is to be used as a reference manual. For more information, please refer to the following supplementary material:

- Monroe Utility Programs Programmer's Reference Manual
- Monroe Operating System Programmer's Reference Manual
- Monroe PASCAL Programmer's Reference Manual

# SECTION 2
# STARTING UP

## SECTION 2
## STARTING UP

### 2.1 LOADING AND STARTING

The editor is delivered to you as a task-file on a 5" disk. Once the bootloading process has been completed, the editor may be called.

#### Logging In

The program is started by the command:

```
EDIT[,,addmem]<fdl>[,fd2]
```

The [,,addmem] is optional and will expand the edit-buffer. It is specified as an integer in number of bytes.

The <fdl> is the name of the source file or the device that contains or shall contain the text. It is specified in the form of a file descriptor.

The optional [fd2] is the name of the destination file or the device where the result should be placed. If no [fd2] is specified, the resulting text is placed in the file specified in <fdl>.

#### Specifing the File Descriptor

As noted in section 1.4 of this manual, the file descriptor, fd, can be composed of four fields: voln, filename, directory, and type.

There are several ways to call the editor:

| | |
|---|---|
| EDIT fd | Creates a new text file or edits an existing file. Input and output files are the same. A backup file is created. |

| | |
|---|---|
| EDIT fd1, fd2 | Creates a new text file by editing an existing file. This method can be used to simply make a copy of a file. No backup is created. |
| EDIT fd,PR: | Outputs the text file to the printer. If corrections were just made, they will appear in the listing. The original file, however, remains unchanged. This means that any text modification or generation performed during this editing session will not be output to the specified file. |

Examples:

| | |
|---|---|
| EDIT GAMES | This allows the user to create a new text file called GAMES, or edit the file GAMES that already exists in the file directory. |
| EDIT GAMES,SPACE | Creates a new text file SPACE, by editing the existing file GAMES. |
| EDIT GAMES,PR: | Outputs the text file GAMES to the printer along with any editing changes just made. The original file, however, remains unchanged. |

Disk File Procedures

When editing a file, there are several internal procedures with which the user should be aware. First, the editor opens a temporary file with the name EDITTEMPX, where X is a sequence number. Secondly, the contents of the file to be edited must be read into the edit-buffer. Since the editor does not automatically read the text from the file into the edit-buffer, the user must do so with the RE command. The edited text is outputted with the OR or EN command into the temporary file and with the WR command to the printer.

If the file to be edited is larger than the edit-buffer (this condition exists if the message EOF (end of file) and the number of the last line of text in the edit-buffer is not displayed on the console when the RE command is executed) the next part of the file must be read in if further editing is needed. This can be accomplished by using the OR command.

The editing session is completed with the EN command. This results in the following: the temporary file becomes the new text file, and the old text file (the file before editing changes) becomes a backup file. The backup file retains the name of the original file, along with an ampersand, "&". Any previous backup file is deleted.

Example:

original file: GAMES
temporary file: EDITTMPO
backup file: GAMES&

Prompting

Text is edited according to user commands. When the editor expects a command, the symbol ">" is outputted as a prompt to the user. To enter a command, the user types the desired command and terminates with a carriage return. The editor interprets this command and performs the specific operation. When execution of the operation is complete, the prompt character is again displayed.

## 2.2 TERMINAL KEYS THAT CONTROL LINES OF TEXT

The editor is used in conjunction with a terminal device like a CRT. Keys that control lines of text are as follows:

TAB key or CTRL-I = tabulates cursor according to preset columns.
CTRL-H = backspaces 1 position.
CTRL-A = stops multiple line presentation or output on printer.

## 2.3 MODES OF WORK

The editor works in two control modes:

command mode - the editor is ready to accept commands.
inline mode - allows user to insert lines of text.

The command mode is prompted with ">", and is automatically invoked at start up.

The inline mode is prompted with "#". When inline mode is active, lines entered at the terminal are treated as text to be inserted into the output file.

The inline mode is made active by the IL command.

Each of the editor commands has a mode associated with it, as is further explained in Section 4 of this reference manual.

Example:

Insert line(s) of text after the last line in the edit-buffer by:

| | |
|---|---|
| >IL¶<br>#| Inline mode. Key in line(s) of text. Enter text by pressing carriage return. |

Insert line(s) of text after any line in the edit-buffer by:

| | |
|---|---|
| >IL line no.¶<br>#| Inline mode. Insert lines of text after line number indicated. Carriage return. |

The inline mode is terminated by inputting # and carriage return directly after the "#" prompt as follows:

| | |
|---|---|
| #---<br>##¶<br>> | Inserted text is entered by pressing carriage return. Input # and carriage return. The prompt ">" signifies that the command mode is now active. |

# SECTION 3
# USING THE EDITOR

## SECTION 3
## USING THE EDITOR

### 3.1 GENERAL

This section gives to the user an understanding of the capabilities of the editor, as well as the procedures necessary to utilize those capabilities. If possible, this section should be followed while at the terminal.

The editor is logged in by:

```
EDIT fd
```

The editor responds with:

```
LINEORIENTED VERSION n
>
```

The first line informs the user that the editor is operative; n stands for the revision number of the editor program. The prompt character ">" indicates that the editor is in the command mode and is ready to accept command from the user.

A command is a brief phrase which tells the editor what to do. A command is entered following the prompt charcter, and is terminated by a carriage return. The carriage return causes the command to be interpreted by the editor.

### 3.2 TEXT FORMAT

The maximum number of characters allowed in each line of text is 80 characters. If the number of characters exceeds 80 the error message, "LINE TOO LONG", will be displayed.

### 3.3 TEXT DISPLAY

The execution of the command PR causes text to be displayed on the console. Since it is possible that not all the text in the file may be displayed on the console at one time, any remaining text may be displayed by pressing the space bar. If only certain lines of text are of interest the user may specify a range of lines as the operand of the PR command which, when executed, will display only those lines specified.

## 3.4 SETTING THE ENVIRONMENT

Tab control causes text to be aligned at pre-set positions, and thus frees the user from constantly having to space text a desired number of columns.

The BT command allows the user to set tabs. The editor moves the cursor according to the pre-set positions whenever the CTRL-I key is depressed.

## 3.5 TEXT IDENTIFICATION

In order to perform a given editing operation, an editor command usually requires that line(s) of text or a portion of a line be specified. A line of text is a sequence of characters terminated by a carriage return. Text is identified for an operation by the use of editor-assigned line numbers, or by a "string" of characters within a line.

### Line Numbers

Line numbers provide a conventient means for identifying lines of text during an editing session. They are assigned when a file is opened for editing, or are generated when text is entered from the terminal. Each line retains its assigned number as long as the line exists during the editing session. Line numbers are output only when text is displayed on the terminal; they are not written to the output file.

When an existing file is opened for editing each line of text entered is assigned a line number beginning with 1, and incremented by one. The largest line number allowed is 9999.

When text is entered at the terminal, line numbers are assigned according to the command used. For example, the IL command can be used to insert new text into existing text at a specified position. Line numbers for inserted lines have a decimal point following the integer number. Decimal points are generated by incrementing the line number by .01 after each line of new text entered. Thus, the line number of each inserted line is .01 greater than the previous line.

"String" Identification

A line of text can also be identified by specifying a "string", which is a sequence of characters contained within the line. If the specified string occurs more than once in the text being edited, each occurrence of the string is assumed to be the line being searched. Searching for a particular string always begins at the first line in the edit-buffer and proceeds sequentially until the last line in the buffer is reached.

## 3.6 CREATE TEXT

Suppose you wish to create a file. After calling the editor, choosing a filename in the form of a file descriptor, the terminal has the following display:

```
EDIT fd
LINEORIENTED VERSION n
>
```

After the prompt you must open the file with the RE command:

```
>RE¶
```

The editor is now ready to receive commands for text generation. Enter the command IL. This puts the editor in the inline mode. Text may now be entered.

```
>IL
     1.#
```

The "#" prompts for the line of text to be entered. For example, suppose you enter:

```
>IL
     1.# THE EDITOR FACILITATES¶
```

The editor responds after your carriage return with:

```
>IL
     1.# THE EDITOR FACILITATES
     2.#
```

The first line is now accepted and the editor is waiting to receive the next line to be entered.

Now you enter:

```
     2.# SOURCE FILES OF ANY KIND¶
```

In expectation of the next line the editor outputs:

```
     3.#
```

Let us assume that you do not wish to enter any more lines. Type in # and press carriage return. The editor is back in the command mode and outputs the prompt ">".

The terminal will now show:

```
EDIT fd
LINEORIENTED VERSION n
>RE
>IL
     1.# THE EDITOR FACILITATES
     2.# SOURCE FILES OF ANY KIND
     3.##
>
```

Verify the text just entered with the PR command. This command tells the editor to output all lines of text to the terminal. If the text has been entered correctly and the editing session is over, the EN command will save the contents of the edit buffer under the filename specified in your file descriptor. A backup file is also created and both files are entered in your system library for future access.

## 3.7 MODIFY TEXT

Should you decide to modify the existing text you simply specify the appropriate file descriptor when calling the editor and use the RE command to read the contents of the file into the edit buffer.

To add a new line of text after the last line in the file, you use the IL command without specifying an operand. The text entered shall follow immediately after the last line of text in the file. For example, you enter:

```
>IL¶
```

The editor responds with the prompt "#" followed by a line number one greater than the last line of text. You type in:

```
          3.# THE CRT IS USED AS A WORK AREA
```

Display the current text to verify the result:

```
>PR¶
          1. THE EDITOR FACILITATES
          2. SOURCE FILES OF ANY KIND
          3. THE CRT IS USED AS A WORKAREA
>
```

Suppose you wish to insert a line of text between two existing lines. There are two ways to accomplish this. You can either enter the number of the lines after which you want the new line to follow, as the operand of the IL command, or enter as the operand a number which numerically lies between the line numbers of the two existing lines in which you want the new line to be inserted.

The latter number may have up to two digits following the decimal point.

For example, suppose you wanted to insert "GENERATION OF" between lines 1 and 2. You enter:

```
>IL 1¶
```

The terminal now looks like this:

```
>IL 1
     1.01#
```

You enter:

```
     1.01# GENERATION OF¶
```

OR

You could have initially entered:

```
>IL 1.50¶
```

The terminal displays:

```
>IL 1.50
     1.50 #
```

And then the correct text is entered at line 1.50. No matter which method is used, when the text is verified, it should look like this:

```
     PR¶
          1.    THE EDITOR FACILITATES
(1.01 or 1.50)GENERATION OF
          2.    SOURCE FILES OF ANY KIND
          3.    THE CRT IS USED AS A WORKAREA
```

Hence, a new line can be positioned anywhere within existing text by entering the appropriate line number as the operand of the IL command.

Suppose you now want to modify one of the existing lines. This is accomplished by entering the command ED along with the number of the line you wish to change.

For example, if you wanted to add "CORRECTION, UPDATING AND" to the first line of text, you would enter:

```
>ED 1¶
```

The editor responds with:

```
    1. THE EDITOR FACILITATES
   >1. ?
```

You can change the line, add to the line, or just modify a portion of it. The TAB key allows the user to position the cursor at the location of the word or character he wishes to edit without changing any text prior to that location. In this example, the TAB key is depressed until the cursor is positioned one space after the "S", and the following is typed in:

```
CORRECTION, UPDATING AND¶
```

The carriage return enters the new text and the PR command displays the following:

```
     >PR
          1.   THE EDITOR FACILITATES CORRECTION, UPDATING AND
(1.01 or 1.50)GENERATION OF
          2.   SOURCE FILES OF ANY KIND
          3.   THE CRT IS USED AS A WORKAREA
     >
```

Lines of text may also be deleted from the file, and the names of variables may be changed. For further information on commands that modify text, see Section 4 of this reference manual.

### 3.8 SAVE TEXT ON DISK FILE

When the editing session is finished, you will probably wish to save the text just entered so it will be available for future reference. This is accomplished by using the EN command. This command will write the text into the file specified by the file descriptor used when calling the editor. The complete procedure is as follows:

```
EDIT¶
LINEORIENTED EDITOR VERSION
>RE¶
>IL¶ (User enters text.)
     1.    THE EDITOR FACILITATES CORRECTION, UPDATING AND¶
     1.50  GENERATION OF¶
     2.    SOURCE FILES OF ANY KIND¶
     3.    THE CRT IS USED AS A WORKAREA¶
>EN¶
```

The EN command indicates to the editor that the session is over, writes the contents of the edit-buffer into the specified file, closes the file and turns control over to the operating system. The operating system responds with a "ready" message.

Note: The line numbers that were used during the editing session will not be output to the file. When the file is again opened for editing, the editor will assign integers in successive order for the new line numbers.

### 3.9 GET EXISTING TEXT FROM DISK FILE

In order to demonstrate how to edit an existing file, let us use the file just saved. The editor is called using the appropriate file descriptor. The RE command opens the file, deletes old text from the edit-buffer and reads the contents of the file just specified into the edit-buffer.

The first thing you might do is display the lines:

```
>PR¶
     1.  THE EDITOR FACILITATES CORRECTION, UPDATING AND
     2.  GENERATION OF
     3.  SOURCE FILES OF ANY KIND
     4.  THE CRT IS USED AS A WORKAREA
```

Suppose you want to add new lines at the end of the text. The IL command without an operand allows you to do so. The editor responds, and you type in the following:

```
>IL¶
     5.# WHERE THE USER MAY VIEW HIS FILE¶
     6.# AND MODIFY, REARRANGE OR DELETE IT¶
```

Now display the text:

```
>PR¶
     1.  THE EDITOR FACILITATES CORRECT, UPDATING AND
     2.  GENERATION OF
     3.  SOURCE FILES OF ANY KIND
     4.  THE CRT IS USED AS A WORKAREA
     5.  WHERE THE USER MAY VIEW HIS FILE
     6.  AND MODIFY, REARRANGE OR DELETE IT
>
```

To save all the text enter:

```
>EN¶
```

All six lines are output to the file specified when the editor was called. The old file is now the backup file; any previous backup file is deleted.

## 3.10 EDITING TIPS

### Working with Large Volumes

The size of the edit-buffer depends upon the available memory in your system. Your text files may be larger than the edit-buffer. If the RE comand is executed and the EOF message is not displayed on the terminal, such a condition has arisen. The remaining portion of the text file is edited, then, in increments the size of the edit-buffer.

This can be accomplished by using one or more of the following commands:

| | |
|---|---|
| RE | Reads the next portion of the file into the edit-buffer. Check if EOF is displayed. |
| OR | Outputs the current edit-buffer, deletes its contents, and reads the next portion of the file into the buffer. |
| WR | Outputs the current edit-buffer but does not delete its contents. Thus, it is possible to duplicate any text in the edit-buffer. |
| AB | Aborts the editing session. The original text file is left unchanged. |

Note: Only the RE and OR commands move text from the original file into the edit-buffer.

### Saving Work Periodically

When heavily editing large files or generating large amounts of text, it is a good idea to periodically save the edited text and resume a new session. In the event that the system crashes, there is always a fairly up-to-date version of the file available.

It is also advisable to save a copy of your file on a different diskette. If the original diskette becomes lost or damaged, a copy of the file is still available.

Renumbering

Often during editing, many lines are inserted throughout the text. In order to make the line numbers less cumbersome to use, the user may renumber the lines of text. The NU command sequentially renumbers all the lines of text in the edit-buffer.

# SECTION 4
# DESCRIPTION OF COMMANDS

# SECTION 4
# DESCRIPTION OF COMMANDS

## 4.1 GENERAL

The editor commands are divided into three groups. The first group deals with those commands that manipulate and display the text. These may require an operand which specifies a range of lines upon which the command operates. The second group of commands controls Input/Output procedures, and the third group of commands controls the environment in which the editor operates.

The general command format is:

    MNEMONIC [operand1],[operand2]

More than one blank may separate the command mnemonic from the first operand. Each successive operand may have leading blanks.

Operands are separated by a comma, or in some cases, by a minus sign. A command is terminated by a carriage return. Only one command may be entered per line.

## 4.2 NOTES ABOUT COMMAND DESCRIPTIONS

An operand of a command is the element(s) upon which the command operates. The following gives detailed definitions of each of the operands that may be required by several of the editor's commands:

### String Operand

| | |
|---|---|
| <string> | specifies a string operand.<br>A string is a sequence of one or more alphanumeric characters. |

## Line Number Operand

line no. Takes the form of:

integer

or

integer.decimal part

where

$0 \geq \text{integer} \leq 999$

$0 \geq \text{decimal part} \geq 99$

The decimal part of a line number cannot exceed two digits.

## Examples:

1. Valid line number

32
1
9999
6.99
1900.6
.11

2. Invalid line numbers

| | |
|---|---|
| 5.678 | -too many decimal places |
| 69001 | -integer too large; maximum is 9999 |
| 2. | -missing decimal digit |
| -15 | -negative number |
| .6 | -missing integer part |

## 4.3 COMMANDS WHICH MANIPULATE AND DISPLAY TEXT

These commands are described in detail in the following pages. The commands in this group are:

| | |
|---|---|
| CV | -Change Variable |
| DL | -Delete Line |
| ED | -Edit Line |
| IL | -Insert Line |
| line no. | -Insert, Replace or Delete Line |
| PR | -Print Text on Console |
| SV | -Search Variable |

## Change Variable (CV) Command

Function: Replaces every occurence of $string_1$ with $string_2$.

Mode: Command.

Format: CV <$string_1$>,<$string_2$>

Arguments: $string_1$ is the string that is to be replaced.
$string_2$ is the string that replaces $string_1$.

Use: Allows the user to change symbols throughout the text with single command.

Note: If operand $string_2$ is omitted, all occurrences of $string_1$ will be deleted from the edit-buffer.

Example: Ex. 1

Existing Text:

```
10. REPEAT
11.    IF RSLT <>0 THEN
12.    ANS:=TRUE
13. UNTIL SUM:=CNTR
```

Enter Command: >CV TRUE,FALSE¶

Resulting Text:

```
10. REPEAT
11.    IF RSLT <>0 THEN
12.    ANS:=FALSE
13. UNTIL SUM:=CNTR
```

Ex. 2

Existing Text:

```
101. CALL  TEST
-
202. CALL  TEST
-
303. CALL  TEST
```

Enter Command: >CV TEST,TESTNUM¶

Resulting Text:

```
101. CALL  TESTNUM
-
202. CALL  TESTNUM
-
303. CALL  TESTNUM
```

## Delete Line (DL) Command

**Function:** Deletes line(s) of text specified.

**Mode:** Command.

**Format:** DL <arguments>

**Arguments:** Can be specified as the number of a line to be deleted or as a range of lines. If a range of lines are specified, $L_1 - L_2$, then all lines between and including $L_1$ and $L_2$ are deleted.

**Use:** The command allows the user to remove the text no longer needed in the file.

**Example:** Ex. 1

Existing Text:

```
21. WHILE RECNO >= 0 DO
22.    BEGIN
23.    READLN(FILID);
24.    IF FILID >= 0 THEN
25. SEEK (FILID,RECNO);
```

Enter Command: >DL 23¶

Resulting Text:

```
21. WHILE RECNO >= 0 DO
22.    BEGIN
24.    IF FILID >= 0 THEN
25. SEEK(FILID,RECNO);
```

Ex. 2

Existing Text:

```
155. STRUCTUR=
156.   RECORD
157.     NAME,COMPANY:STRING[32];
158.     STREET:STRING[20];
159.     CITYSTATE:STRING[30];
160.     TEL:STRING[10];
```

Enter Command: >DL 156-158¶

Resulting Text:

```
155. STRUCTURE=
159.   CITYSTATE:STRING[30];
160.   TEL:STRING[10];
```

## Edit Line (ED) Command

| | |
|---|---|
| Function: | Edits characters within a line. |
| Mode: | Command. |
| Format: | ED <line no.> |
| Arguments: | <line no.> is the number of the line to be edited. |
| Use: | Allows user to make corrections on a line of text. |
| Notes: | The correction procedure is controlled by the following control keys: |

| | |
|---|---|
| TAB key: | Displays the next character in the specified line. This key moves the cursor character by character along the line so that the user may modify only those character(s) he chooses. |
| CTRL-H: or BACKSPACE key | Deletes the last character displayed, moving the cursor to the left. |
| ESC key: | Aborts the ED-command. The original contents of the line, prior to the command, is received. |

Example:

Ex. 1

Existing Text:
```
31. THE FLOPPY DISK CAN BE
32. DAMAGED IF
33. NOT HANDLED CAREFULLY.
```

Enter Command: >ED 32¶

The Editor Responds With:
```
32. DAMAGED IF
32.?_
```

Depress the TAB key to move cursor to the right, displaying the contents of the line character by character, until the cursor is at the desired location.

```
32. DAMAGED IF
32.?DAMAGED_
```

Now type in "OR RUINED".
The console should now look like this:

```
32. DAMAGED IF
32.?DAMAGED OR RUINED IF
```

Since the TAB key positioned the cursor before the "IF" to insert "OR RUINED" the "IF" is pushed over. By depressing the TAB key once again, the "IF" is displayed following the new text. A carriage return enters the line and the editor executes the command.

## Insert Line (IL) Command

Function: Lines of text are read from the terminal and are inserted after the last line in the edit-buffer, or after the line specified in the command.

Mode: Inline.

Format: IL [line no.]

Arguments: [line no.] is the number of the line after which the new line is to be inserted.

Use: Allows the user to insert lines of text anywhere in the file.

Note: Once the IL command is entered, the editor responds with the prompt "#" which indicates that the editor is ready to accept text. Each line of text entered is terminated by a carriage return and the editor responds with a new line number and prompt.

When the user is finished entering text, the mode may be changed by keying in "#" followed by a carriage return.

If carriage return is entered immediately on a new line, the line is stored as an empty line. This causes compiler or assembly error if not deleted. In order to skip a line between lines of text, depress the space bar until the cursor is located at the end of the line and then enter the line. This causes the ASCII representation of of the null character to be entered as the contents of the line. Thus, there is no assembly or compiler error.

Example: Ex. 1

Existing Text:

```
55. DISK FILES ARE BEING USED
56. ALMOST UNIVERSALLY IN SMALL
57. GENERAL PURPOSE COMPUTERS
58. DISK STORAGE DEVICES AVAILABLE
59. RANGE FROM THE SMALLEST OF
60. THE MINI FLOPPY DISKS
61. TO THE LARGE MULTIDRIVE HARD
62. DISK SYSTEMS
```

Enter Command: >IL 60¶

Editor's Response: 60.01 #

Type In: CAPABLE OF STORING 90 K BYTES¶

Resulting Text:

```
55.    DISK FILES ARE BEING USED
56.    ALMOST UNIVERSALLY IN SMALL
57.    GENERAL PURPOSE COMPUTERS
58.    DISK STORAGE DEVICES AVAILABLE
59.    RANGE FROM THE SMALLEST OF
60.    THE MINI FLOPPY DISKS
60.01  CAPABLE OF STORING 90K BYTES
61.    TO THE LARGE MULTIDRIVE HARD
62.    DISK SYSTEMS
```

Ex. 2

Existing Text:

```
55.   DISK FILES ARE BEING USED
56.   ALMOST UNIVERSALLY IN SMALL
57.   GENERAL PURPOSE COMPUTERS
58.   DISK STORAGE DEVICES AVAILABLE
59.   RANGE FROM THE SMALLEST OF
60.   THE MINI FLOPPY DISKS
60.01 CAPABLE OF STORING 90K BYTES
61.   TO THE LARGE MULTIDRIVE HARD
62.   DISK SYSTEMS
```

Enter Command: >IL¶

Editors Response: 63.#

Type In: THAT CAN STORE MILLIONS OF BYTES.¶

Resulting Text:

```
55.   DISK FILES ARE BEING USED
56.   ALMOST UNIVERSALLY IN SMALL
57.   GENERAL PURPOSE COMPUTERS
58.   DISK STORAGE DEVICES AVAILABLE
59.   RANGE FROM THE SMALLEST OF
60.   THE MINI FLOPPY DISKS
60.01 CAPABLE OF STORING 90K BYTES
61.   TO THE LARGE MULTIDRIVE HARD
62.   DISK SYSTEMS
63.   THAT CAN STORE MILLIONS OF BYTES
```

## Line Number Command

**Function:** Deletes, replaces, or inserts a line using the line number specified.

**Mode:** Command.

**Format:** Line no. [string]

**Arguments:** Line no. is the number of the existing line you want to delete or replace, or the number of a line to be inserted within the existing text. [String] is the sequence of characters that is to replace the existing text in the line of text specified.

**Use:** Allows the user to quickly replace, delete or insert a line by specifying only a line number as the command mnemonic.

**Note:** By specifying the number of a line of text that exists, the user can replace or delete the line. If the line no. is followed by a [string], the content of the specified line is replaced with the [string]. If no text is entered after the line no., the line is deleted.

If the user specifies a line number that lies between two existing lines of text, the line is inserted accordingly; its content is specified by the [string].

**Example:** Ex. 1

Existing Text:

```
121. TOTAL:=TOTAL+GRADES(I);
122. AVG:=TOTAL/COUNT;
123. IF AVG > TEMP THEN
124.   TEMP:= AVG;
```

Enter Command: >121.50 COUNT:=COUNT+1¶

Resulting Text:
```
121.    TOTAL:=TOTAL+GRADES(I)
121.50  COUNT:=COUNT+1;
122.    AVG:= TOTAL/COUNT;
123.    IF AVG > TEMP THEN
124.    TEMP:= AVG,
```

Ex. 2

Enter Command: >121 TOTAL:=TOTAL+EXAMS(I)¶

Resulting Text:
```
121.    TOTAL:=TOTAL+EXAMS(I);
121.50  COUNT:=COUNT+1;
122.    AVG:=TOTAL/COUNT;
123.    IF AVG > TEMP THEN
124       TEMP:= AVG;
```

Ex. 3

Enter Command: >122.¶

Resulting Text:
```
121.    TOTAL:=TOTAL+EXAMS(I);
121.50  COUNT:=COUNT+1;
123.    IF AVG > TEMP THEN
124.      TEMP:=AUG;
```

## Print (PR) Command

**Function:** Displays text currently in the edit-buffer on the console.

**Mode:** Command.

**Format:** PR [arguments]

**Arguments:** Arguments can take the form of a range of lines to be displayed or a number of a line to be output to the console. If a range of lines is specified, $L_1$-$L_2$, all lines between and including $L_1$ and $L_2$ are displayed. If no arguments are specified, the current content of the edit buffer is displayed.

**Use:** Allows the user to display lines in order to verify text modification and generation.

**Note:** If the number of lines to be displayed exceeds the number that may appear on the console at one time, the space bar may be used to display any or all of the remaining lines of text.

**Example:**

Ex. 1

Existing Text:

```
89. BEGIN
90.     RESET(FIN,'OLDFILE');
91.     REWRITE(FOUT,'NEWFILE');
92.     WHILE NOTE EOF(FIN) DO
93.       BEGIN
94.         RECNUM:=RECNUM+1;
95.         WRITELN(RECNUM);
```

Enter Command:>PR 92¶

Console Display:92. WHILE NOT EOF(FIN) DO

Ex. 2

Enter Command:>PR 90-94¶

```
Console Display:90. RESET(FIN,'OLDFILE');
                91. REWRITE(FOUT,'NEWFILE');
                92. WHILE NOT EOF(FIN) DO
                93.   BEGIN
                94.     RECNUM:=RECNUM+1;
```

Ex. 3

Enter Command:>PR¶

```
Console Display:89. BEGIN
                90.     RESET(FIN,'OLDFILE');
                91.     REWRITE(FOUT,'NEWFILE');
                92.     WHILE NOTE EOF(FIN)DO
                93.       BEGIN
                94.         RECNUM:=RECNUM+1;
                95.         WRITELN(RECNUM);
```

## Search Variable (SV) Command

**Function:** Searches for all occurrences of the string specified.

**Mode:** Command.

**Format:** SV <string>

**Arguments:** The <string> is a sequence of one or more ASCII characters surrounded by any valid delimiter, such as blanks, parentheses, period, comma, etc.

**Use:** Allows the user to locate variable names, numbers, commands, etc.

**Note:** No more than one space may separate the command mnemonic from the argument.

**Example:**

Ex. 1

Existing Text:

```
132. READLN(EXP);
133. IF LENGTH(EXP)>0 THEN
134.     REPLY:=TRUE
135.     ELSE TEMP:= EXP;
```

Enter Command: >SV EXP¶

Result: 132. 133. 135.

Ex. 2

```
Existing Text:  47. IF RECNUM > 0 AND ≤ 9999 THEN
                48.    BEGIN
                49.      FID:=FILE(RECNUM);
                50.       SUM:=SUM+1;
                51.       WRITELN('NEWFILE',FID);

Enter Command:>SV sum¶

Result:         50.   50.
```

## 4.4 COMMANDS WHICH CONTROL I/O

These commands control the inputting and outputting of text from files into the edit-buffer, and from the edit-buffer into new or existing files. The commands in this group are listed below and are described in detail in the following pages.

| Command | Function |
|---|---|
| OR | Output and Read |
| RE | Read |
| WR | Write |

## Output and Read (OR) Command

Function: Outputs the contents of the edit-buffer to the temporary file. The edit-buffer is deleted and the next part of the source file is read into the buffer.

Mode: Command.

Format: OR

Arguments: None.

Use: Facilitates the outputting of the edit buffer, its deletion and the reading of the next part of the source file into the edit-buffer in a single command.

Example: Enter Command:

```
>RE¶
      -    (editing the first
      -     part of the file)
      -
>OR¶
      -    (output and delete the edit-
      -    buffer; read next part of
      -    the file into buffer and
      -    continue editing)
>OR¶
      -
      -    (etc.)
      -
>EN¶       (end editing session)
```

## Read (RE) Command

**Function:** Reads the contents of the source file into the edit-buffer; opens a new file.

**Mode:** Command.

**Format:** RE

**Arguments:** None.

**Use:** The command allows the user to open a new file, and to read the contents of an existing file into the edit-buffer at the start of an editing session.

**Note:** The execution of two RE commands before an EN or AB command deletes the source file. The backup file remains unchanged. In order to read the next part of the source file into the edit-buffer (after using the RE command initially) use the OR command.

**Example:**

Ex. 1

Enter Command:

| | |
|---|---|
| >RE¶ | The EOF message indicates that |
| EOF | complete file is in the edit- |
| 95. | buffer; 95 is the number of |
| > | the last line of the text in |
| | the file |

Ex. 2

Enter Command:

| | |
|---|---|
| >RE¶ | The first 200 lines of the |
| 200. | file have been read into |
| > | the edit-buffer. |
| - | |
| - | |
| - | |

| | |
|---|---|
| >OR¶ | Outputs the first part of the file (200 lines in this example) to the temporary file. The next part of the file is read into the edit buffer. |
| - | |
| - | Editing session (2). |
| - | |
| >OR¶ | The second part of the file is output to the temporary file and a third portion of the file is read into the edit-buffer. |
| etc. | |
| >EN¶ | End of session. |

## Write (WR) Command

**Function:** Writes the contents of the edit-buffer to the specified output destination. the contents of the edit-buffer, however, are not deleted; the editing session continues with the same text.

**Mode:** Command.

**Format:** WR

**Arguments:** None.

**Use:** Allows the user to write edited material to the specified destination without losing the current contents of the edit-buffer.

**Note:** This command should be used when the destination specified is the printer and the user wants more than one copy of this file. If the source file is specified as the output destination, multiple copies of the edit-buffer-- one for each WR executed-- will be written to the file.

## 4.5 COMMANDS WHICH CONTROL THE ENVIRONMENT

These commands control the environment in which the editor operates. This includes setting tabs and renumbering lines of text as well as the ability to delete the edit-buffer and end the edit session. The commands in this group are listed below and are described in detail in the following pages.

| Command | Function |
|---|---|
| AB | Abort |
| EN | End Edit Session |
| KI | Kill Buffer |
| NU | Renumber |
| BT | Set Tabs |

Abort (AB) Command

Function: Aborts the editing session.

Mode: Command.

Format: AB

Arguments: None.

Use: Allows the user to abort the editing session, transferring control to the operating system.

Note: When the AB command is executed, any text generation or modification performed in the previous editing session is lost. Hence, the source file remains unchanged.

## End Edit Session (EN) Command

Function: Completes the editing session; the temporary file becomes the new text file, or if an existing file was edited, the temporary file becomes the new edition of the original text file. The original file becomes the backup file. Control is then returned to the operating system.

Mode: Command.

Format: EN

Arguments: None.

Use: Allows the user to save on a disk file any newly generated text or text modification once editing is complete.

## Kill Buffer (KI) Command

Function: Deletes the contents of the edit-buffer so that new text may be entered; the source file will contain the new text.

Mode: Command.

Format: KI

Arguments: None.

Use: Allows the user to delete the contents of a file in the edit-buffer.

Note: The KI command deletes the contents of the edit-buffer. Remember, however, that when the EN command is eventually executed, the contents of the edit-buffer have been output to a temporary file and this temporary file becomes the new version of the original file. Thus, the source file no longer contains the original text, but rather contains the text entered following the execution of the KI command.

## Renumber (NU) Command

**Function:** Sequentially renumbers lines of text in the edit-buffer that have been and inserted during an editing session.

**Mode:** Command.

**Format:** NU

**Arguments:** None.

**Use:** Allows the user to Renumber the lines of text in the edit-buffer so that further editing is less cumbersome.

**Example:**

Existing Text:

```
 49.    LD    H,CURSPT
 50.    LR    L,H
 50.01  ADR   H,D
 50.02  POP   D
 53.    LR    A,B
 55.    ST    A,(H)
 56.    PUSH  C
 60.    LA    B,TEMP1
100.    CALL  BINTREE
```

Enter Command:

```
>NU¶
```

Result:

```
 49.    LD    H,CURSPT
 50.    LR    L,H
 51.    ADR   H,D
 52.    POP   D
 53.    LR    A,B
 54.    ST    A,(H)
 55.    PUSH  C
 56.    LA    B,TEMP1
 57.    CALL  BINTREE
```

## Set Tabs (BT) Command

Function: Formats text according to user-specified values.

Mode: Command.

Format: BT <C1>,[C2,...C7]

Arguments: C1...C7 are the values of the tabulation columns. Their values must be an integer between 1 and 80.

Use: Allows the user to preset tabulation points so that subsequent tabbing while editing is not necessary.

Note: The default tabulation values when the editor is invoked are columns 11, 17 and 35, which are used by the assembler.

The CTRL-I key positions the cursor according to the preset column values.

Example: Enter Commands:

```
>BT  10,35,60¶
>IL¶
```

Enter Text:

```
1. #CTRL-INAMECTRL-ICOMPANYCTRL-IST
2. #CTRL-ISMITH,L.CTRL-IBOSECTRL-I MAIN
3. #CTRL-ICOLE,W.CTRL-IMOBILCTRL-IHIGH
4. #CTRL-IBROWN,K.CTRL-IRCACTRL-IPARK
5. ##¶
```

Enter Command: >PR¶

Resulting Text:

| | | | |
|---|---|---|---|
| 1. | NAME | COMPANY | ST |
| 2. | SMITH, J. | BOSE | MAIN |
| 3. | COLE, W. | MOBIL | HIGH |
| 4. | BROWN, K. | RCA | MAPLE |

# APPENDIX A
# COMMAND SUMMARY

## APPENDIX A
## COMMAND SUMMARY

| Command | Format | Function |
|---|---|---|
| Abort | AB | Aborts editing session. |
| Change Variable | CV <string1>,<string2> | Replaces every occurrence of string1 with string2. |
| Delete Line(s) | DL<line no. or $L_1$-$L_2$> | Deletes line(s) of text specified. |
| Edit Line | ED <line no.> | Edits characters within a line. |
| End Edit Session | EN | Ends the edit session. |
| Insert Line | IL [line no.] | Inserts lines of text after last line in edit-buffer, or after line specified. |
| Kill Buffer | KI . | Kills the contents the edit-buffer. |
| Line Number | Line no. | Deletes, replaces or inserts a line. |
| Output and Read | OR | Outputs edit-buffer and reads in next part of the file. |
| Print | PR[line no. or $L_1$-$L_2$] | Displays lines of text on the console. |
| Read | RE | Reads the file into the edit-buffer; opens a new file. |

| Command | Format | Function |
|---|---|---|
| Renumber | NU | Renumbers lines of text in the edit-buffer. |
| Search Variable | SV <string> | Searches for all the occurrences of the string specified. |
| Set Tabs | BT <cl, c2, c3> | Sets tabulation columns. |
| Write | WR | Writes the contents of the edit-buffer to the printer. |

# APPENDIX B
# ERROR MESSAGES

# APPENDIX B
# ERROR MESSAGES

Messages are output to the console when one or more of the following conditions occur:

- -The syntax of the command is incorrect.
- -The editor cannot complete the execution of the command as specified.

## Error Messages From The Editor

| | |
|---|---|
| LINE TOO LONG | -The user has entered a line that exceeds 80 characters; the line is not accepted. The editor switches to the command mode. |
| BAD COMMAND | -The user has entered a non-existent command. |
| SYNTAX ERROR | -One or more operands are missing. |
| CAN'T FIND | -Editor's response to the SV command when the specified string is not found. |
| RENAME ERROR | -System error; you have probably lost your file. |
| END OF MEMORY | -While in the Inline mode, text inserted has used up all available memory. |

# INDEX

INDEX

READER COMMENT FORM

DATE ___________

Your comments and suggestions help to improve this publication.
Please complete the questionaire. Fold, staple, and mail it to Monroe.

Name ______________________ Title ______________________

Organization ______________________

Street ______________________ State __________ Zip __________

Publication Title ______________________

Publication No. __________ Revision Letter __________ Date __________

CIRCLE YOUR RESPONSES TO THE STATEMENTS BELOW. IF YOU RESPOND "NO" TO A STATEMENT, ENTER THE STATEMENT NUMBER AND THE PAGE AND PARAGRAPH IN THE PUBLICATION THAT PROMPTED YOUR RESPONSE.

1. The publication was used for
   - Learning
   - Reference
   - Sales
   - Installing
   - Maintaining
   - Programming

2. The user/reader was
   - High-level Programmer
   - Occasional Programmer
   - Student Programmer
   - Data Entry Operator
   - Other (specify) __________

3. The material is accurate. YES NO

4. The material is clear. YES NO

5. The material is complete. YES NO

6. The material is well organized. YES NO

ENTER DETAILED INFORMATION FOR STATEMENTS 3-6.

| Statement No. | Page No. | Paragraph No. | Comments |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

7. The overall rating for this publication is

   Very Good    Good    Fair    Poor    Very Poor

   Briefly explain your rating. ______________________

8. Additional comments ______________________

FOLD

Software Publications Dept.

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL
FIRST CLASS • PERMIT NO. 731 MORRISTOWN, N. J.

POSTAGE WILL BE PAID BY ADDRESSEE

MONROE division of
Litton Business Systems, Inc.
Box 9000R
Morristown, N.J. 07960

CUT ALONG LINE

FOLD

STAPLE

STAPLE