COMPUTER ONE PASCAL

for the Sinclair QL Computer

A USER GUIDE

TABLE OF CONTENTS

Appendices

Computer One welcomes ideas and comments. These and any bug reports or further enquiries should be sent on the report form at the back of this manual to:

Technical Enquiries, Computer One Limited, Science Park, Milton Road, Cambridge CB4 4BH.

CHAPTER ONE
INTRODUCTION

This user guide provides the information required to edit, run and compile programs using the Computer One Pascal system on the Sinclair QL microcomputer. The guide is not intended to teach Pascal and the reader is referred to the Bibliography for a number of references on learning Pascal.

The Computer One Pascal system is a powerful menu driven system which allows you to edit, compile and run programs with ease – there are no command names to remember as all commands are executed by selecting from the displayed menu. In addition, a help window aids you whenever data input is required. The system editor is a general purpose screen editor, which has been designed with usefulness and simplicity in mind. You are not restricted to editing files used by the Pascal system and can use the editor to edit any QL text files. The editor also allows you to edit Pascal source files while examining compilation errors for the file.

As well as editing, compiling and running Pascal programs, you also have the option of formatting and examining microdrive cartridges, using the format and directory commands, and deleting and copying microdrive files. There is also a Make Job command, which makes a compiled Pascal program into an EXECable QDOS job which can run outside the Pascal system.

The Computer One Pascal compiler produces a compact intermediate code which is interpreted when the program is run. Intermediate code is generally more compact than equivalent 68000 code.

The language implemented is close to the ISO Pascal standard specification, with a small number of differences. The language is described in Chapter Five.

A number of extra procedures and functions are provided to allow you to make full use of the QL from the Pascal system. These procedures allow you to use the QL windowing and graphics features and also to make calls to machine code routines.

1.1 USING THIS GUIDE

This section contains a brief overview of the contents of each chapter in the guide.

CHAPTER 2 describes how to get started using the Pascal system and works through an example of an edit, compile, run session.

CHAPTER 3 describes the Pascal system environment and gives a description of each of the commands.

CHAPTER 4 describes, in detail, the general purpose editor, including the edit listing option, which allows you to edit a file while looking at the error messages produced for that file by the compiler.

CHAPTER 5 describes the Computer One version of the Pascal language.

CHAPTER 6 describes the standard procedures and functions which have been added to the language to allow you to make use of the QL.

CHAPTER 7 gives a brief description of the standard Pascal procedures and functions.

Appendices – The appendices list the compiler, run-time and system error messages, give the full syntax of Computer One Pascal, and give some example programs.

CHAPTER TWO

GETTING STARTED

This chapter describes how to use the Computer One Pascal system and works through an example of editing, compiling and running a short Pascal program.

2.1 BACKING UP

The supplied microdrive cartridge should be backed up immediately on receipt. This cartridge should be treated as a Master copy. It is recommended that you make two backup copies using the Master cartridge only as an emergency backup and not to run the software. Backing up may be done by running the supplied 'CLONE' program as follows:

1. Place the Master copy in microdrive 2 (the right hand side drive).

2. Place a blank cartridge in microdrive 1.

3. Enter the following command:

```
    LRUN mdv2_clone <ENTER>
```

4. The QL will respond with various instructions to name the new cartridge and initiate the copying. MAKE SURE THE MASTER IS IN DRIVE 2.

5. The 'cloned' system may be used as soon as the microdrives have stopped running.

Repeat the procedure with another cartridge, and store the master and one of the copies in a safe place. Use the remaining copy as your working master – only use the others in an emergency. Please note that you may only copy the software for your own use.

2.2 STARTUP

The boot file on the supplied Pascal microdrive cartridge has been set up to load the system from microdrive one and to use microdrive two as the default device when file names are required. You can easily change the boot file so that the system will load from any other device and will use another device as the system default. Whichever device the system is initially loaded from, it will expect all system files, such as the editor and the compiler, to be on this device.

To change the boot device and default device you need to alter the string variables bootdevice$ and defaultdevice$ in the boot file. For example to boot from floppy drive flp1_ and also use flp1_ as the default device you should set bootdevice$ and defaultdevice$ to the string 'flp1_'.

To startup the Pascal system you should load and run the boot file from the required load device, unless you are loading from the QL's default device, in which case the system will automatically be loaded when F1 or F2 is pressed after resetting the system.

The Pascal interpreter, and the environment code are loaded into memory on startup and remain resident while the Pascal system is being used. These are kept resident so that the system does not have to reload the interpreter code from an external device everytime you wish to run a program. The resident code, the screen map and QDOS work space require about 60K of memory, leaving between 65K and 70K in which to run programs, assuming there is no add-on memory. When the editor is loaded into memory there is approximately 50K to 55K of free space for editing the file. The compiler is not resident and is loaded into memory from the load device when you wish to compile a program.

When the system is loaded, a system command menu will appear on the screen, waiting for you to select commands to be run. To select an item use the cursor up (↑) and cursor down (↓) keys or enter the number of the command you wish to select. The currently selected menu item is always highlighted. When a command has been selected it can be run by pressing the ENTER key.

The window at the bottom of the screen tells you how to select and run menu items. Help messages will appear in this window to guide you when a command is waiting for input from the keyboard.

If you choose the EXIT menu option the system will return to BASIC and the system can then be rerun by inserting the Pascal cartridge or disc into the load device and typing 'Pascal'. This only applies if you have not reset the QL since you last exited the Pascal system.

2.3 EXAMPLE SESSION

A number of example programs are included on the system cartridge and we shall now work through one of the example programs showing how to edit, compile and run it. The Pascal microdrive cartridge should still be in drive one. The file containing the program is called 'fib_pas'. It is recommended practice to use meaningful extensions for file names, so that it is easy to tell what type of file it is. For example 'fib_dat' might be data file and 'fib_pas' might be a Pascal source file. For this reason the compiler insists that Pascal text files should have the extension '_pas'.

The example program generates the first n terms of the Fibonacci sequence, where each term, beyond the first two, is generated from the sum of its two nearest predecessors, the first two terms being 0 and 1. Thus the sequence starts :–

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

Before we attempt to run the file we will take a look at it using the editor. Select the editor command by pressing '1' or using the cursor control keys. Now press ENTER to run the editor. The main menu screen will disappear and a new screen will appear with another menu. This is the Editor File Menu. Use the cursor keys to select the 'load file' option. When this option has been selected press ENTER to use this option. In the window along the top of the screen the prompt

```
Load which file :
```

will appear. You should now enter

```
mdv1_fib_pas
```

We shall assume in this example that the system has been loaded from microdrive one and that the default device is microdrive two. If you wish to specify a file on drive two there is no need to give the drive name, but to specify a file on any other drive the name must be given.

The following Pascal program will now appear in the editing window :–

```
program fibonacci ;
var fib1, fib2, i, n : integer;
begin
   {generate each number from the sum of its two
   predecessors}
   fib1 := 0 ;
   fib2 := 1 ;
   i := 2;    { number of fib numbers generated }
   writeln ('Generate how many Fibonacci numbers ?');
   readln ( n ) ;
   while i < n do
   begin
       writeln ( fib1, fib2 ) ;
       fib1 := fib1 + fib2 ;
       fib2 := fib1 + fib2 ;
       i := i + 2 ;
   end ;
   if i = n then writeln ( fib1, fib2 )
   else writeln ( fib1 )
end.
```

The program does not need to be changed, so when you understand how it works you can exit the editor. Press F1 to get the editor file menu to reappear. Use the cursor keys to select the 'leave' option. When this has been selected press ENTER and the System Command Menu will reappear. We now want to select the 'Compile' option. When you have selected this option press ENTER. The following prompt will appear on the screen next to the menu :–

```
Which file ? [_pas]
```

The bracketed '_pas' indicates that the compiler will automatically append this extension to the filename you enter. Notice that the help message at the bottom of the screen has now changed to tell you that you can press ENTER to return to the menu and that mdv2_ is the default microdrive. Enter the filename now :–

```
mdv1_fib
```

Next you are prompted for the output file name :–

```
Output file [_qlp]?
```

We shall make the output name fib_qlp so you should enter

```
mdv1_fib
```

The compiler always produces a file with the extension '_qlp' (**QL** Pascal) and again you need not specify the file extension when entering the name.

When you have done this the compiler will be loaded and a new screen will then appear. There is a window along the bottom of the screen, which the compiler uses to output messages, for example if the compiler could not find a particular file on microdrive, a message would be output in this window. The compiler does not produce a full listing to the screen, unless the listing option is switched on, but only outputs lines with errors and the appropriate error numbers. The error messages can be displayed and the file re-edited using the 'edit listing' option in the editor. This is described in section 4.8. Unless you have changed the file 'fib_pas' there should be no errors and the compiler will output a message indicating that the file has been successfully compiled. A message will then be displayed asking you to press the escape key to return to the menu – this allows you time to examine any errors you might have.

Having compiled the file and returned to the menu we can now try running the program. Select the 'run' option and press ENTER. You will be given the following prompt :–

```
Which file ? [_qlp]
```

Enter the file name

```
mdv1_fib
```

Next you are prompted for the size of the stack to use. For the moment just press ENTER to get the default stack size. The file 'fib_qlp' will now be loaded, a clear screen will appear and the program will run. This program will prompt for a number n and then print to the screen the first n numbers in the Fibonacci sequence. When the program has finished you will be asked to press the escape key to return to the menu screen.

A more detailed description of all the menu commands can be found in Chapter 3 of this user guide.

CHAPTER THREE
THE COMMANDS

When the Pascal system has been loaded, the System Command Menu is displayed. This screen allows you to select commands from the menu and run them, and to input the data required by some commands when they execute. The screen has three main windows :–

1) The MENU window displays the names of the seven available commands. The cursor up (↑) and cursor down (↓) keys or the numeric keys are used to select menu items, the current item always being highlighted. A command is run by selecting it and pressing the ENTER key.

2) The CONSOLE window, to the right of the menu window, is used by some commands to prompt for data and to output data. Error messages are also displayed in this window.

3) The HELP window, along the bottom of the screen, is used to give information about what input the system currently expects.

There are currently nine commands available on the system. These are the edit, compile, run, directory, delete, copy, format, make job and exit commands. Several of the commands prompt for filenames and the following section describes what are valid file names.

3.1 FILENAMES

A valid file name can have up to 36 characters, including a 4 character extension [_aaa], but not including the device name [e.g. mdvx_]. The system does not differentiate between upper and lower case letters and they can be used interchangeably. If the prompt for a file name is followed by an extension in brackets, the system will expect a file name with no extension and will automatically append the displayed extension to the name. Alternatively a name with the displayed extension can be entered.

If a file name entered is too long the error message

```
*** Error – bad name
```

is output and the system returns to the command menu.

Note that the default device name for the system need not be added to file names on the default device. The system will first attempt to find a file with the exact name entered, including the required extension if there is one. If this file cannot be found, the name is prefixed with the default device name and the system attempts to find this file. The supplied Pascal microdrive cartridge has the default device set to 'mdv2_'. However by altering the boot file you can change this (see section 2.2).

When you are promted for a file name you can specify a device name, for example 'ser1' or 'scr_'. This allows files to copied to a printer, the screen or any other device.

3.2 THE COMMANDS

The following sections give a description of each of the commands.

3.2.1 Edit

When this command is selected the screen is cleared and the editor is run. The operation of the editor is described in detail in Chapter 4.

3.2.2 Compile

When the compile command is selected the console window displays the prompt

```
Which file ? [_pas]
```

and the help message at the bottom of the screen changes to remind you of the default device name and that ENTER can be pressed to return to the command menu. The system will attempt to find the given file, with the '_pas' extension.

If it is not found the error message

```
*** Error – not found
```

is displayed and the system returns to the command menu. Otherwise the prompt

Output file [_qlp] ?

is displayed and you should enter a name for the code file to be produced.

Again ENTER can be pressed to return to the menu. The extension '_qlp' is added to the name if it is not specified. Note that the code file may be produced on a device other than that of the source file. Next the compiler is loaded from the load device and a new screen appears. At the top is a title window and at the bottom is a message window which is used for messages and prompts. The Pascal system cartridge or disc should always be kept in the drive from which it was loaded, as the compiler needs to access the device periodically.

The compiler does not output a full listing of the file being compiled to the screen, unless the listing option is turned on (see 3.2.2.1), but displays lines with errors and the appropriate error numbers. In the message window is a prompt giving you the option of quitting the compilation or of continuing. If no key has been pressed after thirty seconds the compiler will automatically continue the compilation. An example of a compilation error is shown below.

```
        for i = 1 to 10 do
                ↑
51 Press ESCAPE to quit, any other key to continue
```

An error indicator appears under the symbol on the source line which caused the error. The error indicators appear on the same line, unless two errors occur at the same symbol, in which case the second indicator for the symbol and subsequent indicators appear on the next line. The line after the error indicator line(s) contains the error numbers. These are provided for convenience, since the error messages can be examined in the source code using the 'edit listing' facility in the editor (see section 4.8). The compiler produces a file with the extension '_err', which is used by the editor to display the error messages. This option is particularly useful if you have a number of compilation error messages. The compiler will halt after twenty errors have been output. A full list of the compiler error messages and corresponding error numbers can be found in Appendix A.

If the ESCAPE key is pressed the compilation terminates, otherwise it continues and further errors, if they occur, are output in the same manner. When the compilation has finished it outputs the number of errors found. You are then asked to press the escape key to return to the Command Menu.

Note that the ESCAPE key can be pressed at any time during a compilation if you wish to stop the compilation.

3.2.2.1 Compiler Options

There are five options available in the compiler. These are

1) **The listing option** (l) – if this is turned on at any point in the program, a compilation listing of that part of the program is produced on the screen. The listing is not saved on the microdrive cartridge. If the listing option is turned off only lines with errors and the appropriate error numbers are displayed. The default listing option is off. This option tends to slow down the compilation.

2) **The code option** (c) – if this is turned on, the intermediate code file is produced. If it is turned off at any point in the program no code is produced for that part of the program. There is little point in turning off the code production, except at the start of the program, in which case no code will then be produced and the compilation will run more quickly. If the code option is turned off at any point after the start of the program, the code file will be deleted by the compiler. This option is useful while compilation errors are being removed from a program. The default code option is on.

3) **The range-check option** (r) – if turned on, this option produces code that checks all array indices are within the array bounds and that all subrange values are within the the declared range. It is recommended that this option is turned on during program development, since values outside the range will cause run-time error messages, rather than unpredictable results. The default option is on.

4) **The nil-pointer check** (n) – if turned on, this option produces code that checks that no attempt is made to access a variable through a **nil** pointer. It is recommended that this option is turned on during program development, since accessing variables through **nil** pointers will cause run-time error messages, rather than unpredictable results. The default option is on.

5) **The trace option** (t) – if turned, on this option produces code that causes the name of a procedure to be written to the standard output file whenever it is called. If turned off again only the names of those procedures which are between the option being turned on and off in the source file will be output. The default for this option is off.

Compiler options are written in the program file as comments, with a dollar sign immediately following the opening comment character. The options are signified by a letter and a plus or minus sign to indicate whether the option is to be turned on or off. Thus the default for the five options would be inserted at the top of the program file as

```
{$l-,c+,r+,n+,t-}
```

Note that there must be no spaces between the specified options.

3.2.3 Run

When the run command is selected the console window displays the prompt

```
Which file ? [_qlp]
```

The system will attempt to find the given file, with the '_qlp' extension, on microdrive. If it is not found the error message

```
*** Error – not found
```

is displayed and the system returns to the command menu. If the file is found a prompt is made for the stack size required to run the file. If you press ENTER the default stack size of 10K is used. This should be sufficient for most applications. See section 3.2.8.1. The system will then load the file, clear the screen and then run the program. If the interpreter detects an error while interpreting the program, an error message is output and the user is asked to press any key to return to the System Command Menu. An example of an error during execution of a program is an array bounds error, where the program uses an array index which is outside the dimensions of the array. A full list of run-time error messages and their possible causes is given in Appendix B.

When a run-time error occurs, a listing of the nested procedure and function calls is output to assist in debugging, the most recently called procedures and functions appearing at the top of the list. The name at the top of the list is the procedure or function in which the error occurred.

3.2.4 Directory

When this command is selected the following prompt is displayed on the console window :–

```
Which drive [1..8] ?
```

If you enter any number between 1 and 8 the system will attempt to read the directory of the cartridge or disc in the given drive (8 is the maximum number of microdrives on one QL). If you press 'o' the prompt

```
Which device ?
```

is displayed and you can enter a device name. Any characters, other than 1..8 and 'o' are ignored and the system will continue to wait for a numeric key or 'o' to be pressed. If there is no cartridge or disc in the given drive or if the specified drive or device does not exist, the error message

```
*** Error – not found
```

is displayed. Otherwise the name and length of each file on the cartridge or device are displayed as shown below.

```
fib_pas – 834 bytes
fib_qlp – 510 bytes
letter_txt – 4352 bytes
fib_err – 35 bytes
```

The system then returns to the Command Menu. Note that the CTRL and F5 keys can be pressed together to stop a window scrolling. Press any key to start it scrolling again.

3.2.5 Delete

When this option is selected the console window displays the prompt

```
Which file ?
```

Note that the full file name must be specified for this command. If the given file is not found the error message

```
*** Error – not found
```

is output. Otherwise the system will display the file name and ask for confirmation of the deletion :–

```
Hit Y to confirm deletion >
```

If either the 'Y' or 'y' keys are pressed the file will be deleted and the system will return to the Command Menu, otherwise the system returns immediately to the Command Menu.

3.2.6 Copy

When this option is selected the console screen shows the prompt

```
Which file ?
```

Note that the full file name must be specified for this command. If the given file is not found the error message

```
*** Error – not found
```

is output. Otherwise the prompt

```
To which file ?
```

is displayed. The new file name should be specified in full. If the file to be copied to already exists, the name of the file and the message

```
File exists. Overwrite ?
```

are displayed and if you press a 'Y' or 'y' the file will be overwritten, otherwise you will return to the Command Menu. When the file has been copied the system returns to the Command Menu.

3.2.7 Format

This command allows you to format a microdrive cartridge or other storage medium. When the command is entered the prompt

```
Which device ?
```

is displayed and you should enter the device and medium name, for example 'mdv2_testfiles'. You will then be asked press 'y' to confirm the format.

3.2.8 Make Job

This command allows you to make a compiled Pascal program into a QDOS job which can run outside the Pascal system. Usually you would only use this facility when you have completely debugged a program. When the command is entered the prompt

```
Which file [_qlp] ?
```

is displayed. If the given file does not exist the error message

```
*** Error – not found
```

is output and the system returns to the command menu. Otherwise you are prompted for the output file name, which has the extension '_exe'

```
Output file [_exe] ?
```

When you have entered the output file name you are prompted for the job stack size. This is the amount of data space which will be used when the job is running. Section 3.2.8.1 gives some guidelines for working out the stack space required by the Pascal program. If you just press ENTER the default stack size of 10K is used. Note that the system must have access to the device from which the system was loaded when this command is used, as a number of the system files are required.

Note that if you run a QDOS job using the EXEC command you will have to use CTRL-C before the program can get input from the keyboard, and when the program has finished you will have to use CTRL-C again to return to another QDOS job or to BASIC.

3.2.8.1 Guidelines for stack sizes

It is necessary to specify the stack size when making a Pascal program into a QDOS job so that the job can be given that amount of data space when it is EXECed. If there is plenty of space available on the machine when the program is being run then giving a large stack size is unimportant, but if you want the job to be as small as possible then it is desirable to keep the data space to a minimum, while making sure that the stack is not going to overflow when the program is running.

The system also prompts for the stack size when a Pascal program is run within the Pascal system. This allows you to experiment with the stack sizes while developing a program. If you are not running very large or highly recursive programs, then the default size of 10K should be sufficient and you can just press ENTER when prompted. If you are interested in optimising the stack size then the following paragraphs are intended to give some help in estimating the amount of stack required.

Whenever a procedure call is made the procedure parameters and all the data declared in the procedure have to be put on the stack. In addition a certain amount of housekeeping information is required on each procedure call. The housekeeping information and the data for a procedure is called a frame. If the procedure calls are nested, then there must be enough room on the stack for all the frames of the nested procedure calls. Space is also required for the global data declared at the outer level of the program. When a program is compiled you are informed of the data space required by the data of each procedure and at the end of the compilation the data space for the main program is the global data space required. The housekeeping information for a stack frame requires a maximum of 24 bytes. ThePascal interpreter needs about 200 bytes of stack for its own use and if real numbers are being used an additional 200 bytes is required. The data sizes, in bytes, of the standard types is shown below:

integer, subrange, enumerated type, pointer – 4;
boolean, char – 1; real – 6; set – 16;
record – sum of size of components;
file – 8 + size of base type;
array – number of elements * size of base type;

Example: A program with one procedure requires 4K for its global data and 1K for the data declared in the procedure. If the procedure is called recursively to a depth of 3 then the stack space required is

$$200 + 4K + 3\star(1K + 24) = 200 + 4096 + 3144 = 7440 \text{ bytes}$$

It is obviously better to give too much stack than too little, although the system will produce a run-time out of memory error if the stack overflows. In this case a sufficient stack size would be 7500 bytes. Note that if you enter a stack size of less than 512 bytes the stack size will be set to 512.

3.2.9 Exit

When this command is entered the screen will be cleared and the system will return to BASIC.

CHAPTER FOUR

THE EDITOR

The editor has been designed with ease of use in mind. It makes use of the cursor control keys, situated on either side of the space bar, and the function keys on the left hand side of the QL keyboard. It also provides menus similar to the system command menu, to allow you to select file saving and loading options easily and to copy and delete pieces of text.

4.1 EDIT FILE FORMAT

Edit files consist of lines of printable characters, the lines being separated by the line feed character. These files are called text files. Although the editor only allows you to create text files it does not check, when loading a file, that the file only contains printable characters and line feeds. However, you are strongly recommended to edit only text files.

Edit lines can only be as long as the edit window. However there is a special character '<<', which, if it occurs as the first character on a line, will be treated as a concatenation character. When a file is loaded, any lines longer than the edit window will automatically be split, with the concatenation character being inserted at the split. When a file is saved, a line will be joined to the previous line if the concatenation character appears at the beginning of the line. If this character is deleted it causes a permanent split in the line.

4.2 EDIT FILE NAMES

The editor will allow any file name to be specified (see section 3.1 for valid file names) except a name with the extension '_err' or '_qlp', since names with these extensions have a special meaning in the Pascal system. If, when the editor is prompting for a file name, an invalid name is given or a file error occurs (for example when attempting to load a file, a file with the given file name must exist), an error message is displayed in the help/error window and the file name prompt is redisplayed.

4.3 EDITING A FILE

When the editor is invoked from the command menu a new screen appears with four windows – an edit window, in which the editing of text takes place; a prompt window, in which the editor displays prompts and receives input (for example a file name); a help/error window, in which error and help messages are displayed; a lift window – the lift facility is described in section 4.7. A menu will then appear on the screen, allowing you to choose the file you wish to edit or to choose to edit a new file. This menu, called the Editor File Menu, is described in section 4.4.

When a file has been loaded the edit screen will appear with the first few lines of the file. The cursor, a solid rectangle, will appear over the first character of the file. If a new file is being edited the edit window will be blank, except for the cursor which will be at the top left corner. The cursor represents the current editing position in the file and all editing operations take place at the cursor position. Whether a new file is being edited or an existing file is being edited, the operation of editing the file is exactly the same and is described in the following sections.

4.3.1 Moving the cursor

The cursor can be moved anywhere in the file by using the four cursor control keys which are situated on either side of the space bar on the QL keyboard. Each of the keys moves the cursor one character position in the direction shown on the key. There are several special cases which affect the movement of the cursor :

1) If the cursor is moved right when it is at the end of a line it moves to the start of the next line.

2) If the cursor is moved left when it is at the beginning of a line it moves to the end of the previous line.

3) If the cursor is moved up or down and the previous or next line is shorter than the current one the cursor moves to the end of the new line.

4) If the cursor moves off the top or bottom of the screen the text is scrolled up or down by one line.

5) The cursor remains stationary if any attempt is made to move it off the beginning or end of the file.

If you use the ALT key with either the cursor left or cursor right keys, the cursor moves to the beginning or the end of the current line.

4.3.1.1 Paging

If the SHIFT key and the cursor up or cursor down keys are pressed together, the window will scroll down or up by the number of lines that will fit in the window. The cursor is set to the middle of the window.

4.3.2 Inserting Text

To insert text simply start typing characters at the keyboard. Each character is inserted at the cursor position, and the character at the cursor position and all characters to the right are shifted right by one character position. When the cursor is the last character on a line and a new line is required press the ENTER key and a blank line will be inserted below the current line, with the cursor at the start of the new line. The editor has an auto indent facility which sets the cursor under the first non-blank character of the previous line when a new line is taken.

Lines can only be as long as the window and any attempt to insert a character which will make the line too long, will cause the QL to emit a beep and the character will not be inserted. However the end of line character can be inserted and if you wish to have the line longer than the edit window, you can continue the line by inserting the special character '<<' at the beginning of the next line. This character is keyed in by pressing the CTRL, SHIFT and X keys together. The editor will automatically remove this character and the preceding end of line character when the file is saved.

4.3.3 Deleting Text

The character to the left of the cursor can be deleted by pressing the CTRL key and the cursor left key together. The cursor and all characters to the right are shifted left by one character position. This is the opposite of inserting a character and is useful for correcting typing errors as text is being entered.

The character under the cursor can be deleted by pressing the CTRL key and the cursor right key together. The cursor remains in the same position and all characters to the right of the cursor are shifted left by one character position.

If the CTRL, ALT and cursor left(or right) keys are pressed together all characters from the cursor position to the start (or end) of the line are deleted.

4.3.4 Splitting and concatenating lines

A line is split by positioning the cursor at the position in the line where the split is required and pressing ENTER. This causes the character under the cursor and all characters to the right to be inserted below the line.

Two lines are concatenated by deleting the line feed at the end of the first of the two lines. The line feed is always the last character in any line and is represented as a blank. The line feed is deleted in the same way as any other character, using the CTRL and cursor right keys if the cursor is over the line feed, or using the CTRL and cursor left keys if the cursor is at the start of the next line. When the two lines are concatenated all lines below will be scrolled up by one line. If concatenating the two lines will make the resulting line longer than the edit window the QL emits a beep and the line feed character is not deleted. However the two lines can be concatenated when the file is saved, by inserting the concatenation character '<<' at the beginning of the second line. This character is entered by pressing the CTRL, SHIFT and X keys together.

4.3.5 Finishing the edit session

When you have finished editing the file, or wish to save the file and continue editing the same file or another file, the function key F1 can be pressed to select the Editor File Menu which allows the file to be saved. The Editor File Menu is described in the next section.

4.4 THE EDITOR FILE MENU

This menu appears on the screen when the editor is run from the Pascal command menu, or when the function key F1 is pressed. When the menu is displayed it appears with a number of options. Each option is numbered and is selected by pressing the required number or using the cursor up and cursor down keys to step through the options. The currently selected option is always highlighted. When the required option has been selected the ENTER key should be pressed to use the option. In certain cases some of the options are unavailable, for example the 'save' options cannot be used when no file is currently being edited. When an option is unavailable it is displayed in red (options are normally displayed in white) and cannot be selected using the numeric or cursor keys. You can press the F1 key again if you want to leave the File Menu and return to the edit window. When you are prompted for a file name you can just press ENTER to return to the menu.

Each of the options is described below.

4.4.1 Create New File

If no file is currently being edited, the edit window is cleared and you can start editing a new file. If a file is currently being edited and has been altered, you are asked if you wish to save the changes. If you answer yes and the current file is not a new file it is saved, otherwise the prompt

```
save to which file :
```

appears in the message window and the file will be saved with the entered file name.

4.4.2 Load File

If this option is selected and a file is currently being edited the same procedure is carried out as described for the above option 4.4.1. When the current file has been saved, or no file is currently being edited, the message

```
load which file :
```

appears in the message window. The editor then loads the given file and the edit window appears with the first few lines of the file. You can now start editing the file.

If the file name given has the extension '_pas' the editor checks if there is also a file with the same name, but with the extension '_err'. If one exists the editor automatically uses the '_err' file to produce a listing which allows you to see the error messages produced when the specified file was last compiled. The 'Edit listing' facility is described in section 4.8.

4.4.3 Read file

If this option is selected you will be prompted for a filename. A copy of the file will then be inserted at the cursor position.

4.4.4 Save a Copy

This option is only available when a file is currently being edited. The message

```
save to which file :
```

appears in the message window and the file is saved with the given file name. You can then continue editing the same file. Note that saving a copy with a given file name does not affect the name of the file being edited. i.e. the file being edited still has its original file name.

4.4.5 Save and Continue

This option is available only if a file is currently being edited. If an existing file is being edited the file is saved and you can continue editing. If the file being edited is a new file, the message

```
save to which file :
```

is displayed in the message window and the file is saved with the given name. You can then continue to edit the file.

4.4.6 Save Listing

This option is only available if the **edit listing** facility is currently being used.

The listing, the file with the embedded error messages, is saved to a file with the same name as the file being edited, except that the extension '_lis' is used. This option is useful if you wish to print the listing file. When the listing has been saved you can continue to edit the file.

4.4.7 Remove error text

This option is only available if you are editing a listing. It should be used only after you have corrected the errors indicated by the merged error file. When selected the error text is removed and the file no longer is a listing. Removing the error text allows a copy of the file to be saved or the save and continue option to be selected.

4.4.8 Directory

This option allows you to display the contents of a microdrive cartridge or other device. The prompt

```
Which device :
```

appears in prompt window and you can enter any device name, for example 'mdv2_'. The directory, if it exists, is then displayed in the edit window.

4.4.9 Save and Leave

This option is only available if a file is currently being edited. If an existing file is being edited, the file is saved with the existing file name, otherwise, the file is a new file and the message

```
save to which file :
```

appears in the message window and the file is then saved with the given file name.

When the file has been saved the editor is exited and the system returns to the Command Menu.

4.4.10 Leave

This option quits the editor without saving the changes and returns to the Command Menu. If a file is currently being edited the message

```
Lose changes (y/n) :
```

is displayed in the message window and only if you enter 'Y' or 'y' will the editor be exited. Otherwise the edit window will be redisplayed and you can continue to edit the file.

4.5 EDIT MENU

This menu appears when the F5 key is pressed. It has four options

– cut pieces of text from the edit file – copy pieces of text from the edit file – paste pieces of text into the edit file – display the last piece of text cut or copied

To leave the edit menu you press F1. Each of the options are described below. Some options expect a marker to be set. This is described in the following section.

4.5.1 Inserting a marker

A marker is set in the text by pressing the CTRL and F4 keys together. The character at this position is shown in reverse video. Only one position in the file may be marked at one time. The marker will be removed by any action except moving the cursor (either with the cursor keys, paging or the lift). The marked character will only be displayed in inverse video until it moves off the screen.

4.5.2 Cut

This option is only available if a marker has been set. It allows you to remove a piece of text from the edit file and save it in an internal buffer. The contents of this buffer can then be pasted into another part of the file. To cut text, set a marker (CTRL-F4) at the start of the text to be removed. Then place the cursor after the text to be cut. Now press F5 to get the edit menu, select the cut option and press ENTER. The text will be removed from the file but will remain available to be subsequently pasted. The text cut includes the marked character but does not include the character at the cursor.

You will only be allowed to cut if there is sufficient space to act as an internal buffer. In a QL with 128K of memory the buffer space is approximately 3K. In a QL with more memory the buffer size is about one eighth of the available editing space. Any existing text in the internal buffer will be deleted before the cut.

4.5.3 Copy

This is similar to the cut option except that the text selected is copied into the internal buffer without being removed from the edit file.

4.5.4 Paste

This option is only available if there is some text in the internal buffer, that is, if you have cut or copied text. The cursor should be moved to where you wish the text to be pasted. Press F5 to get the edit menu, then select paste and press ENTER. The text in the internal buffer is inserted at the current cursor position unless the resulting file is too big for the workspace in which case an error will occur and the paste will not proceed.

4.5.5 Show buffer

You may examine the contents of the internal buffer by pressing F5 to get the edit menu, selecting the show buffer option and pressing ENTER. The buffer contents are then displayed.

4.6 STRING SEARCH

The string search is initiated when the function key F2 is pressed. It searches for a given string from the current cursor position. When F2 is pressed the message

```
search for which string :
```

is displayed in the message window. You should then enter the required string. If the string is found in the text, the cursor is set to the last character of the string. If the string is not found, the cursor position remains as it was and a beep sounds. If F2 is pressed together with the SHIFT key, a search is made for the string specified in the last search. If, when the prompt for a string is displayed, just the ENTER key is pressed, you will immediately return to the editor.

4.7 EDITOR LIFT

The lift is a means of moving through a file quickly and easily. While in normal edit mode the lift arrow moves up and down as the cursor is moved up and down through the text. The top of the lift window represents the top of the file, the bottom of the lift represents the bottom of the file and the lift arrow represents the cursor position in the file.

When the function key F3 is pressed the editor goes into lift mode. Lift mode allows you to move the lift arrow by using the up and down cursor keys (for extra speed use the ALT key and cursor up or down). When F3 is pressed again the editor returns to the normal edit mode with the cursor in the position implied by the lift arrow. This is most useful for long files where paging many times becomes tiresome.

4.8 EDIT LISTING FACILITY

This facility allows Pascal programs to be edited, while looking at any error messages produced when the program was last compiled. The compiler produces a file with the extension '_err', which contains information about the errors. If you choose to edit a Pascal file (a file with the extension '_pas') and an '_err' file exists for the Pascal file, then the editor automatically uses the '_err' file to produce a listing, which appears in the edit window as a normal file. Below each line of the file which had a compilation error, is a copy of the line and the error messages. Each error message is 'bracketed' by the character '§', which cannot be deleted or inserted in the editor by the user. When the file has been loaded the '_err' file is deleted, since the errors will no longer be valid if the file is changed.

The file may be edited as normal and when it is saved any parts of the file enclosed in the '§'s are not written to the microdrive file. Thus the error messages are removed when the file is saved. If, when saving the file, the **save listing** option is chosen (see 4.4.6) the whole file, including the error messages, are saved to a file with the same name as the Pascal file, but with the extension '_lis'.

CHAPTER FIVE
LANGUAGE DEFINITION

Computer One Pascal is close to the ISO Pascal standard specification, with the following main exceptions :–

1) Procedures and functions may not be passed as parameters.

2) I/O has been expanded and modified slightly, in order to give the user the full benefits of the QL I/O. See section 6.1

3) No checking of assignment to the control variable of a **for** loop is made, either direct or indirect assignment. i.e. whether the variable is actually assigned to in the body of the loop or whether the variable is assigned to in a procedure or function called from the body of the loop.

4) Gotos may not jump out of procedures or functions.

In the following sections describing Pascal the syntax of the part being described is usually given. The full syntax of Computer One Pascal is given in Appendix C.

The notation used within the text for describing the syntax is similar to Backus-Naur Form. An example of a rule is

```
        program = program-heading block '.'
```

This can be read as 'a program consists of a program-heading followed by a block followed by a period'. Special symbols in Pascal, defined in section 5.1, are always enclosed in quotes in a rule. Any entity in a rule enclosed in curly brackets ({}) indicates that the entity is repeated zero or more times.

An entity enclosed in square brackets ([ ]) is optional. If entities are separated by a vertical bar (|), this indicates that one of the entities is selected.

Examples :

```
(1) integer = [sign] unsigned-integer
(2) unsigned-integer = digit { digit }
(3) sign = '+'|'-'
```

These rules can be interpreted as follows :

(1) an integer is an optional sign followed by an unsigned integer
(2) an unsigned-integer is a digit followed by zero or more digits
(3) a sign is either a plus or a minus

5.1 FUNDAMENTALS

The alphabet of Pascal consists of letters, digits and special symbols. Sentences are constructed from the alphabet according to the syntax of Pascal. Pascal does not differentiate between upper and lower case letters, except in strings and comments, and they may be used interchangeably.

The special symbols in Pascal are shown below

```
+ - / * = <> < > <= >= := ( ) [ ] { } . , : .. ↑
of do to or if in not div mod set and
end var nil for then type file with goto
case else until while begin const label array
repeat downto packed record program function
procedure
```

Computer One Pascal contains four additional symbols :

```
bor band bxor bnot
```

The words which are special symbols are reserved words and may not be used for any other purpose.

5.1.1 Numbers

Pascal numbers may be either integers or reals. An integer number is a whole number which may be positive, negative or zero. The number is written as a sequence of digits, optionally preceded by a sign (+ or −). No other characters must appear in an integer. In Computer One Pascal the range of integers is

−2147483647 to +2147483647

A real number, which may be preceded by a sign, may be written in one of two forms.

(1) The number has a decimal point, which must be preceded and followed by at least one digit.

Examples :

```
1.2
-34.789
+0.45609
-0.0007
```

(2) The number is expressed as an integer or decimal number multiplied by an integral power of 10, known as the exponent. The exponent is preceded by the letter 'E' or 'e' and an optional sign.

Examples :

```
2.3E6
54E+5
45.6E-3
0.76E2
```

In Computer One Pascal the range of real numbers is

+/−(1E−615 to 1E+615) with 8 significant digits

5.1.2 Identifiers

Pascal identifiers, which are names used to identify data locations and pieces of program text, consist of a letter followed by any number of letters and digits.

Examples :

```
a
currchar
x3
extremelylongnameindeeed
```

In Computer One Pascal identifiers can be any length, but only the first eight characters are significant. Remember that reserved words cannot be used as identifiers.

5.1.3 String Constants

A Pascal string constant consists of a sequence of characters enclosed in quotes. If a quote itself is required in a string it should be written twice. A string constant of length one is a character constant.

Examples : ‘hello’
‘I don‘‘t know’
‘‘‘’

There are a number of characters which have a special meaning when they are preceded in a string by the single quote character. These characters are

‘n : This represents the character line-feed
‘r : This represents the character carriage return
‘t : This represents the TAB character
‘p : This represents the form-feed character
‘0 : This represents the null character

These characters are useful for formatting output when using the write statement (see 5.6.2).

5.1.4 Comments

A Pascal comment is a sequence of characters enclosed in curly brackets ({ }). Comments may appear anywhere in the text, except within an identifier, number or special symbol, and have no effect on the execution of the program.

Example : { this is a comment ! }

5.2 PASCAL PROGRAMS

Every Pascal program consists of a program header and a block.

program = program-header block ‘.’

The program header gives the program name and the parameters. In Computer One Pascal these parameters need not be specified. They are ignored by the compiler.

A program block consists of a number of declaration parts followed by a statement part.

```
        block = [label-declaration-part]
                  [constant-definition-part]
                  [type-definition-part]
                  [variable-declaration-part]
                  procedure-and-function-definition-part
                  statement part
```

5.2.1 Label declarations

```
        Label-declaration-part = 'label' label {',' label } ';'
```

Any program statement may be marked by prefixing the statement with a label followed by a colon. The label is defined in the declaration part and is an unsigned integer in the closed range 0 to 9999.

5.2.2 Constant definitions

```
        Constant-definition-part = 'const' ident '=' constant ';'
                                            { ident '=' constant ';'}
```

Constant definitions are used to give names to constants. Constant identifiers help to make a program more readable. For example we might define

```
        const     lf = "n';
                  cr = "r';
```

to give names to the ascii control codes line feed and carriage return. In our program we would then use the identifier lf everytime we want to use the control code line feed.

The use of constant declarations is also useful for constant values, which we may at a later date wish to change. For example implementation dependent constants. It is considerably easier to change the definition of a single constant identifier than to search for and change all the occurences of a particular constant in the program.

5.2.3 Type definitions

```
Type-definition-part = 'type' ident '=' type ';'
                                   {ident '=' type ';'}
```

In Pascal data types can be described in the variable declaration part, or may be referenced in the variable declaration part using a type identifier, which has been defined in the type definition part. Section 5.3 contains a detailed discussion of data types, describing what standard types are available and how new data types can be defined.

5.2.4 Variable declarations

```
variable-definition-part =
          'var' ident {',' ident } ':' type ';'
                              { ident {',' ident } ':' type ';' }
```

Example :

```
     var      a, b : integer ;
              c : real ;
              d : array [1..10] of char ;
```

The type may be a standard type, a type description or a type identifier. Every variable used in a Pascal program must be declared in a variable declaration, unless it is predeclared.

Note that a variable may not be declared more than once in the same variable declaration. i.e. the following is illegal

```
     var      x : char;
              x : integer;
```

5.2.5 Procedure and function definitions

Procedures and functions are described in section 5.5

5.2.6 Statements

Statements are described in section 5.4

5.2.7 Scope

In Pascal all declarations have **scope**. The scope of a declaration is the routine in which it is declared, together with any routines declared in that routine and any routines nested deeper within these routines. Declarations in the main program, which are global declarations, have the whole program as their scope.

Outside its scope a name is unknown and the same name can be used in other declarations. If a name is declared within the scope of a declaration with the same name, the inner declaration overrides the outer one. This is known as a **hole** in the scope of the outer declaration.

Example :

```
procedure a ( b : integer; c : real ) ;
{ start of scope of a, b and c }
var d : char ;    { start of scope of d }

    procedure b ;
    { this is a hole in the scope of the integer b }
    var d : char ;
    { this is a hole in the scope of the outer d }
        begin { body of procedure b }
            .
            .
        end ; { procedure b and inner d go out of scope
                here }
    { integer b and outer d back in scope here }
    begin { body of procedure a }
        .
        .
    end ;
    { b, c, d are now out of scope }
```

5.3 DATA TYPES

Data types in Pascal can be highly structured and complex, but are all ultimately built from unstructured types. An unstructured type is either defined by the programmer or is one of the four standard scalar types.

5.3.1 Standard Scalar types

The four predefined scalar types are integer, real, boolean and char.

5.3.1.1 Integer type

A value of type integer is a whole number whose range of values is restricted by the implementation. In Computer One Pascal an integer is 4 bytes long, giving a range of values −2147483647 to +2147483647.

Arithmetic operators which take integer operands and return integer results are :−

+	add
−	subtract
*	multiply
div	whole number division
mod	modulo (remainder after division)

Example :

```
8 div 3 = 2
8 mod 5 = 3
5 - 9 = -4
```

The operators + and − can be used to denote the sign of the operand.

Example :

```
-difference
+2
```

Pascal provides a number of standard functions which may be applied to integers. The standard functions are described in Chapter 7.

A number of the standard functions that relate to integers and other unstructured types, except reals, are introduced below.

The type integer defines an ordered set of values whose range depends on the implementation. Each value has a successor and a predecessor (the least value has no predecessor and largest value has no successor) and such a type is called an **ordinal type**. Pascal provides the functions **succ** and **pred** to give the immediate successor and immediate predecessor of an ordinal type. For integers these functions are equivalent to adding and subtracting 1.

There exists a standard identifier **maxint** whose value is +2147483647 , which can be used for comparison purposes to prevent integer overflow.

5.3.1.2 Real type

A value of type real is an element of a subset of real numbers whose range is implementation dependent. In Computer One Pascal the range of reals is

+/−(1E−615 to 1E+615) with 8 significant digits

Arithmetic operators which take real operands and return real results are :−

+	add
−	subtract
⋆	multiply
/	real division

Example :

21.5 + 2.3 = 23.8
8.0 / 5.0 = 1.6
5.0 − 9.0 = 4.0

The operators + and − can be used to denote the sign of the operand.

Example : −sin (x)
+2.67

Integer types may also be used in real arithmetic. The operators +, − and ⋆ may be used with one real and one integer operand to produce a real result. The operator / may be used with two integer operands to produce a real result. Remember if the two operands are integers an integer result will be produced except when the operator is /.

Examples : 8 / 5 = 1.6
2.5 + 1 = 3.5
4.5 ⋆ 2 = 9.0

As well as the operators described above Pascal provides a number of standard functions which may be applied to reals. The standard functions are described in chapter 7.

Note that real is not an ordinal type and cannot be used with the standard functions **pred** and **succ**. Nor can it be used as an array index or base type of a set.

5.3.1.3 Boolean type

A value of type boolean is one of the logical truth values **true** or **false**. True and false are predefined ordinals such that

```
pred ( true ) = false
succ ( false ) = true
```

5.3.1.4 Char type

A value of type char is an element of the QL character set. The QL character set consists of the ASCII character set, which is represented by the ordinal values 0 to 127, and the QL specific characters, which are represented by the ordinal values 128 to 255. The full QL character set and equivalent ordinal values (ASCII codes) are given in the QL User Manual. Throughout this manual the character set will be referred to as the QL ASCII set.

A value of type char is an ordinal value, the ordering being defined by the QL ASCII codes. Thus

```
succ ( 'A' ) = 'B'
pred ( '0' ) = '/'          since ASCII code for '/' is 1 less than
                            the code for '0'
```

The standard function **ord** is used to map the character set onto a set of non negative integers starting at zero and the function **chr** maps the ordinal number i onto the equivalent character. The set of integers onto which **ord** maps the chacter set is the equivalent QL ASCII code of each character. Similarly **chr** maps a QL ASCII code onto the corresponding character. Thus

```
ord ( '0' ) = 48
ord ( '1' ) = 49 etc.

chr ( 48 ) = '0'
chr ( ord ('0') ) = '0'
```

5.3.2 Enumerated and Subrange types

Enumerated types are user-defined scalar types whose definition indicates an ordered set of values by enumerating the identifiers which denote the values.

```
'type' ident = '(' ident { ',' ident } ')'
```

Example :

```
type colour = ( black, blue, red, magenta, green cyan,
                         yellow, white )
```

The relational operators can be applied to enumerated types, as can the standard procedures pred, succ and ord. Thus using the above example

```
succ ( green ) = cyan
pred ( white ) = yellow
ord ( red ) = 2
ord ( yellow ) = 6
```

Note that the boolean type is equivalent to an enumerated type

```
( false, true )
```

A variable may take a range of values which is a subrange of the value described by some ordinal type. Its type is then defined to be a **subrange** of the host type.

subrange-type = constant '..' constant

The constants must both be of the same ordinal type.

Example :

```
age : 1..100 ;
subcolour : blue..green; { using the above colour
                           definition }
ascii : ''O'..'~' ; { the ASCII character set }
```

Each value of a subrange is taken as being of the host type and operations on the host type can also be applied to operands of the subrange. Thus the functions 'ord', 'succ' and 'pred' can be applied to subranges and 'chr' can be applied to a subrange of type char.

Note that the compiler only allows fifty distinct subranges.

5.3.3 Array Type

array-type = 'array' '[' index-type {','index-type}']' 'of' type

An array consists of a fixed number of components, all of the same type called the **base** type. Each component of the array is accessed by the array variable and an **index**. The type of the base can be any type, the type of the index must be an ordinal type.

Example :

```
var     lengths : array [1..8] of integer ;
        identifier : array [1..20] of char ;
        code : array [letter] of integer ;
        m : array [1..10] of array [1..20] of char ;
```

The lower bound of the array must lie in the range −8388608 to +8388607.

Note that Computer One Pascal does not support packed data structures. The compiler will ignore the word 'packed' when it appears before the word **array**, **record**, **set** or **file**.

When accessing arrays the array index can be an expression, which is of the index type.

Example :

```
        lengths [x mod 8] := y * z ;
        currcode := code [ currletter ] ;
```

where x is of type 'integer' and currletter is of some user defined type 'letter'.

For multi-dimensional arrays the declaration can be abbreviated. For example the array m declared above can be declared as follows

```
        var m : array [1..10,1..20] of char
```

A multi-dimensional array is accessed in the same way as a single dimension array, using a number of indices. For example m could be accessed as follows

```
        m [2,6] := 'a' ;
        m [2] [6] := 'a' ;  { equivalent to above }
        currchar := m [ x, y ] ;
        m [1] [2] := m [3] [2] ;
```

If two arrays are of the same type, (see 5.3.8 on type compatibility) then it is possible to assign the values of one of the arrays to the corresponding elements of the other array in a single assignment statement.

Examples :

```
        var x : array [ 1..10, 1..80 ] of char ;
```

The statement

```
x [1] := x [2] ;
```

copies 80 characters from x [1] to x [2].

```
type    line = array [1..80] of char;
var     thisline : line ;
        fulltext : array [1..100] of line;

fulltext [ currline ] := thisline;
```

In this example **fulltext** is a hundred line array, each line being 80 characters. Since **thisline** is of type **line** and each element of **fulltext** is of type **line** the two are assignment compatible.

5.3.3.1 String arrays

In Computer One Pascal the data structure of type

```
array [1..n] of char
```

is called a string array and has a number of properties which do not apply to other arrays. For example the string

'hello there'

is a constant string array of type array [1..11] of char.

Variables of this type are string array variables of length n. A string array, whether constant or variable is assignment compatible (see 5.3.8.1) with any string array of the same length.

Example : If we have a variable string array declared as

```
var city : array [1..8] of char
```

then we can write the statement

```
city := 'London  ' ;
```

String arrays, unlike other arrays can be used as parameters of write statements.

Example : writeln ('city is ', city)

String arrays of equal length may be compared using the relational operators. These operations use the convention used in dictionary ordering when comparing string arrays.

Examples : If city = 'London ' then ....
'ABC' < 'BCD'
'ABC' < 'ABD'

The above two comparisons are only true if all characters are the same case, either all upper or all lower, since when using the ASCII character set all upper case letters have lower values than the lower case letters. Thus

'A' < 'a'
'Z' < 'a'
'XYZ' < 'abc'

5.3.4 Record Type

It is often convenient to organise a collection of data in which the items are not all of the same type. For this purpose Pascal provides the record. A record is a structure with a fixed number of components, called fields. Unlike an array these fields need not be of the same type.

```
record-type = 'record' field-list 'end' ';'
field-list = [fixed-part [';' variant-part] |
                  variant-part [';']]
fixed-part = ident { ',' ident } ':' type ';'
                  {ident{ ',' ident } ':' type ';' }
```

Example :

```
type person = record
                  name : array [1..20] of char ;
                  sex : ( male, female ) ;
                  age : 1..120 ;
                  end ;
var p : person ;
```

p can be assigned to as follows

```
p.name := 'Gunn Ewan A.A.      ' ;
p.sex := male ;
p.age := 2 ;
```

As with arrays, Computer One Pascal does not support packed records.

In addition to fixed record fields, as shown in the example above, records may also have a variant part, which allows variables to be of the same type, but to differ in structure. Using the above example we might wish to add a field to the record giving the place of birth, if the person is a national, or giving the country of origin and date of entry if the person is foreigner. The record definition would be as follows

```
type origin = ( national, alien ) ;
     person = record
                name : array [1..20] of char;
                sex : ( male, female ) ;
                age : 1..100 ;
                case country : origin of
                   national : (placeofbirth:
                                  array [1..15] of char);
                   alien : ( countryoforign:
                                   array [1..15] of char;
                             dateofentry: record
                                             day : 1..31 ;
                                            year : 70..90;
                                             mnth : 1..12;
                             end);
                end ;
```

The structure of the record is dependent on the value of **country** – if country is set to **national** then **placeofbirth** can be accessed, if **country** is set to **alien** then the two fields of **alien** branch of the variant part can be accessed. The field **country** is known as the **tag field**.

Example :

```
p.name := 'Savedra Sean        ';
p.sex := male ;
p.age := 24 ;
p.country := alien ;
p.countryoforigin := 'A country        ';
p.dateofentry.day := 27 ;
p.dateofentry.year := 65 ;
p.dateofentry.mnth := 6 ;
```

Note that having set the country field to 'alien' it would be illegal, for this particular variable, to attempt to access the 'placeofbirth' field. It is the resposibility of the programmer to ensure that this does not happen.

Only one variant part is allowed in a record and must follow the fixed part of the record. However the variant part of the record may itself have variant parts. If a field of a variant part is empty the form of the field is

lab-list : ()

5.3.5 Set Type

set-type = 'set' 'of' base-type

the base-type must be an ordinal type.

Example :

```
       type months = ( Jan, Feb, Mar, Apr, May, Jun, Jul,
                       Aug, Sep, Oct, Nov, Dec );
             year = set of months ;
```

A set consists of a subset of elements of the base-type. Thus in the above example the set year can consist of any subset of the twelve months, including the empty set and the full set of twelve months. In Computer One Pascal sets can have up to 128 elements, having ordinal values 0..127. Since the QL character values go from 0 to 255, the set, set of char, cannot be used. However the predeclared subrange 'ascii', which is of type char, has ordinal values which are the ascii codes 0 to 127.

Sets are built by specifying their elements, separated by commas and enclosed in square brackets. Expressions appearing in the element specification must all be of the set's base-type.

Example :

[Apr, Jun, Sep, Nov]	denotes a set of type year containing the months having 30 days
['a'..'z', 'A'..'Z']	denotes set of char containing only the alphabetic characters as elements.

The set [] denotes the empty set, the set containing no elements.

The following operators can be used on sets

+	Union
⋆	Intersection
−	Difference
= <>	equality and inequality
<= =>	set inclusion
in	tests if element of base-type is in a set

Examples :

['a'] + ['b']	gives ['a', 'b']
['a','b'] − ['a','c']	gives ['b']
['a','b'] ⋆ ['a','c','d']	gives ['a']
['a'] = ['a','c']	gives false
['a'] <> ['a','b']	gives true
['a'] <= ['a','b']	gives true
['a'] <= ['c','d']	gives false
'a' in ['a', 'b']	gives true

5.3.6 Pointer Type

pointer-type = '↑' type-ident

A variable which is pointed at by a variable of type pointer is a dynamic variable. A static variable is one that is declared in the program and is subsequently denoted by its identifier. It has storage allocated to it during the entire execution of the block in which it has been declared. With dynamic variables storage is allocated by an explicit call, from the program, to the standard procedure **new** and the variable created by **new** is not referenced directly by an identifier, but by a pointer to the variable. Linked lists and trees are easily and efficiently constructed and manipulated using dynamic data structures.

Although a pointer is bound to a particular variable type by its declaration, all pointers can be set to the value **nil** indicating that the pointer does not point to anything. Pointers can be tested for equality with the value **nil**.

Note that for pointer type declarations only, a type identifier may be used before it is declared.

Example :

```
                type      link = ↑ info ;
                          info = record
                                 number : integer ;
                                 next : link ;
                          end ;
```

The field **next** is a pointer which points to a variable of type **info**. Thus we can build up a linked list of variables of type **info**, as shown below

```
var first, ptr : link ;

first := nil ; { initialise to null pointer }
for i := 1 to n do
begin
   new ( ptr );  { create a new var of type info }
   ptr^.number := i; { set the number field of the var }
   ptr^.next := first { link in current var to list }
   first := ptr ;
end ;
```

Note that a pointer that has not been assigned to does not have the value **nil**, but is an uninitialised variable.

Dynamic variables cannot be allocated space in the same way as other variables because the space required is not known until run– time. For this reason Pascal maintains a **heap**. A heap is just an area of free storage, which is used for allocating and releasing the space required by dynamic variables. The space, as mentioned previously, is allocated using the standard procedure **new** and is released using the standard procedure **dispose**.

If we have a pointer to a type t, then the effect of the procedure call

```
new ( p )
```

is to allocate enough space on the heap for a variable of type t and to set p to point to the variable. If t is a record with a variant part (see 5.3.4) the amount of space allocated is enough for the largest variant. If the tag fields of the nested variant-parts are t1..tn then the call

```
new ( p, t1, t2,..., tn )
```

allocates just enough space for the variants specified by the tag values t1..tn. The tag values must be constants and must be contiguous and listed in the order they are declared. The tag values are not allocated to the tag fields by this procedure, but the user should ensure that the correct tag values are allocated when the record is used.

Example :
```
   type origin = ( national, alien ) ;
        person = record
                   name : array [1..8] of char ;
                   age : 1..100 ;
                   sex : ( male, female ) ;
                   case country : origin of
                     national : ( placeofbirth :
                                             array[1..8] of char);
                     alien : ( countryoforigin :
                                             array[1..8] of char);
                              dateofentry : record
                                               day : 1..30;
                                               yr : 70..90;
                                               mth : 1..12;
                                             end );
                 end ;

   var p : ^person ;
```

The **country** field is the tag-field and to allocate just enough space for a national we would use the call

new (p, national)

To release space on the heap use the procedure **dispose** to remove the space allocated for a dynamic variable. Variables must not be referenced after the space allocated for them has been released.

Example :
```
       type    link = ^info ;
               info = record
                         number : integer ;
                         next : link ;
               end ;
       var

          ptr, first : link ;
               .
               .
       first := nil ;
       for i := 1 to n do
       begin
          new ( ptr ) ;
```

```
      ptr^.number := i ;
      ptr^.next := first ;
      first := ptr
   end ;
   { use the linked list for something here }
   { now get back the heap space used for the list }
   { by disposing of each link in the list }
   while ptr <> nil do
   begin
      first := ptr^.next ;
      dispose ( ptr ) ;
      ptr := first
   end ;
```

5.3.7 File types

Since many computer applications involve storage of large volumes of data, which has to be retained from one program execution to another, the data has to be held on some secondary storage device, such as a microdrive.

Any stream of information which is held on some external storage medium for input to, or output by, a computer is called a file. Character devices such as keyboards and VDU can also be considered as files for the input and output of data, although there is no concept of the data being stored.

Pascal allows the user to structure files according to the type of data required.

Example :
```
        type     f1 = file of integer;
                 f2 = file of char ;
                 f3 = file of person;
```

where person is a user defined record.

There is a predeclared type in Computer One Pascal called text, which is a file of char and there are two standard files, input and output, which are the default files for reading and writing.

The use of files in Pascal and Pascal I/O in general are described in detail in Section 5.7.

5.3.8 Type compatibility

Pascal is a strongly typed language. In some contexts two types may be required to be the same, but not in others. Even if two types are not the same, they may still be compatible and in certain contexts this may be sufficient, except in the case of assignments, where the types must be assignment compatible.

Two types must be the same only for variable parameters of procedures and functions (see 5.5.1.1).

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

(1) They are the same type

(2) One is a subrange of the other

(3) Both are subranges of the same host type

(4) Both are set types and have the same base-type

(5) Both are string array types with the same number of elements.

Type compatibility is required in the majority of cases where two or more entities are used together, for example in expressions.

```
Example:        type     t1 = integer;
                         t2 = 1..100;
```

t1 and t2 are compatible since t2 is subrange of t1

5.3.8.1 Assignment Compatibility

The value of an expression 'expv' of type 'expt' is assignment compatible with a variable, a parameter or a function identifier of type 'vt' if any of the following is true (Expressions are described in section 5.4.1.1) :

(1) vt and expt are the same type and neither is a file type and neither is a structured type with a file component.

(2) vt is of type real and expt is of type integer or a subrange of type integer.

(3) vt and expt are compatible ordinal types and expv is within the range of possible values of vt.

(4) vt and expt are compatible set types and all members of expval are within the range of possible values of the base-type.

(5) vt is of type array [1..n] of char and expt is a quoted string constant containing exactly n characters.

5.4 STATEMENTS

The statement part of a program defines the actions to be carried out as a sequence of statements. Statements in Pascal are executed one after another.

```
statement-part = 'begin' statement {';' statement} 'end'
```

5.4.1 Assignment statement

The assignment statement is used to assign a particular value to a variable. The value is specified by means of an expression.

```
assignment-statement =
            ( variable-access | func-ident ) ':=' expression
```

5.4.1.1 Expressions

An expression consists of constant or variable operands, operators and function calls (functions are described in 5.5.2). An expression is a rule for calculating a value, where the rules of operator precedence and left to right evaluation apply. The precedence of operators is given below in decreasing order. Operators of equal precedence are on the same level.

```
not, bnot
*, /, div, mod, and, band
+, -, or, bor, bxor
=, <>, <, >, <=, >=, in
```

The operators **bnot**, **band**, **bor** and **bxor** treat integer values as bit patterns. They all take two integer operands, except **bnot**, which takes a single operand. The table below shows the result of operations using these operators.

operator	**action**
bnot	flip all the bits of the given integer
band	'ands' the corresponding bits of the two integers. Thus only those bits which are set in both numbers are also set in the result.
bor	'ors' the corresponding bits of the two integers, setting all the bits in the result which are set in either of the operands.
bxor	'ors' the corresponding bits of the two integers, setting only those bits in the result that are set in either of the operands but not in both.

An expression in parentheses is evaluated independently of its preceding and succeeding operators.

Examples :	2 ⋆ 8 + 4 ⋆ 2	= (2 ⋆ 8) + (4 ⋆ 2)	= 24
	not a and b	= (not a) and b	
	4 ⋆ 8 / 2	= (4 ⋆ 8) / 2	= 16
	6 + 8 div 4	= 6 + (8 div 4)	= 8
	6 band 12	= 4	
	9 bor 5	= 13	
	9 bxor 5	= 12	

The user must make no assumption about the order of evaluation of operands in boolean expressions.

An assignment can be made to variables of any type except files. However the type of the variable and of the evaluated expression must be assignment compatible (see 5.2.8.1).

Examples :

```
a := 3 + 4 ⋆ 8 ;
s := ['a'..'z'] ; s is of type set of char
ch := 3 ; is illegal if ch is of type char
```

5.4.2 Compound statement

A compound statement is a sequence of statements that are executed in the order in which they are written. The sequence is surrounded by the symbols

begin and **end**. The statement-part of a program is a compound statement.

Example :

```
begin
   a := 3 ;
   b := 5 ;
   c := a * b
end
```

Note that no semi-colon is required after the statement before the **end**, since the semi-colon is a statement separator and is not part of the statement.

5.4.3 Repetion statements

An important class of action in programs is the loop, in which a statement or group of statements executes repeatedly, subject to some terminating condition.

5.4.3.1 While statement

while-statement = 'while' expr 'do' statement

The expression must be of type boolean. The expression is evaluated and if the result is true the statement is executed. This is repeated until the expression evaluates to false, in which case the statement is not executed.

Examples :

```
while x < y do x := f(x) + 1 ;

while ( a <> ' ' ) and ( count <> 10 ) do
begin
    read ( a ) ;
    count := count + 1
end ;
```

Note that if the expression initially evaluates to false, the statement is never executed.

5.4.3.2 Repeat statement

repeat statement = 'repeat' statement {';' statement }
'until' expr

The repeat statment is similar in action to the while statement, but the statement part is executed before the controlling expression is evaluated. Again the expression must be of type boolean. The statements are repeatedly executed until the expression evaluates to true. Since the statement part is executed before the expression, it is always executed at least once.

Examples :

```
repeat a := f(a) + 1 until a > b ;

repeat count := count + 1; read ( c ) until c = ' ' ;
```

5.4.3.3 The for statement

for-stat = 'for' variable ':=' init-expr ('to' | 'downto')
final-expr 'do' statement

The variable must be an ordinal type and the expressions must be of the same type as the variable. The variable must be declared in the innermost block containing the loop.

On entry to the **for** loop the control variable is assigned the value of the initial expression. Each time round the loop the value of the control variable is incremented (decremented in the case of **downto**) until it is greater than (less than) the value of the final expression. The initial and final expressions are only evaluated once, on initial entry to the **for** loop.

If the final expression is less than the initial expression (greater than in the case of **downto**), the statement part is not executed at all. On completion of the **for** loop the value of the control variable is undefined and no assumptions should be made about its value.

Note that Computer One Pascal does not check whether the control variable is assigned to within a **for** loop, although no assignment should be made to the variable.

Examples :
```
        for i := 1 to n do write ( i*i ) ;

        for day := monday to friday do
        begin
           readln ( hoursworked ) ;
           totalhrs := totalhrs + hoursworked
        end
        { assuming day is a user defined ordinal type }

        for x := max downto min do readln ( a[x] ) ;
```

5.4.4 Conditional Statements

It is often necessary to make the execution of a statement dependent on some condition or value. Pascal provides two constructs for this purpose – the if-statement and the case-statement.

5.4.4.1 If statement

This statement has two forms

(1) if-statement = 'if' expr 'then' statement
(2) if-statement = 'if' expr 'then' statement 'else' statement

The expression in both cases must be of type boolean.

If the expression evaluates to true, the statement following the **then** is executed. If it evaluates to false, then in case (1) execution continues at the statement following the if statement, and in case (2) the statement following the **else** is executed.

Examples :
```
        if a < b then a := b ;

        if x + y = z then
        begin x := 0 ; y := 0 end
        else z := 0 ;
```

A nested if-statement of the form

if expr1 then if expr2 then stat1 else stat2 appears to be ambiguous. In Pascal this statement is defined to be equivalent to

```
if expr1 then
begin if expr2 then stat1 else stat2 end
```

i.e. The **else** is associated with the nearest **if**.

Note that there is never a semi-colon before the **else** and that a semi-colon after a **then** would indicate an empty statement.

5.4.4.2 Case statement

The case statement consists of an expression (the selector) and a list of statements, each labelled with a constant of the same type as the the selector, which must be an ordinal.

```
case-stat = 'case' expr 'of'
                        lablist ':' statement
                                    .
                                    .
                                    .
                          'end'
```

The statement selected for execution is the one whose label is equal to the current value of the selector. If none of the labels is equal to the value of the selector an error occurs.

Examples :

```
case i of
   0 : x := 0;
   1 : x := i ;
   2 : x := i * i ;
   3 : x := i * i * i ;
   4 : x := i * i * i * i ;
end;


case currchar of
   lf, cr : newline := true ;

   'a','b','c','d' : letter := true ;

   '0','1','2','3' : digit := true ;
end ;
```

Case labels are not ordinary labels and cannot be used in a goto statement. Also the labels in a case statement must be unique, no label appearing in more than one branch of the statement.

5.4.5 The Goto Statement

goto-stat = 'goto' label

The **goto** statement is used to cause execution to continue at the statement prefixed with the given label. The scope of the label is the entire block in which it was declared. No label can be used to prefix more than one statement.

Note that the effect of jumping into a structured statement from outside the statement is undefined. Further note that Computer One Pascal does not allow jumps from inside a procedure to a statement outside the procedure.

Goto statements should be used with care and only in uncommon or unusual situations, for example when an error is detected.

Example :

```
begin
   .
   .
   if x < 0 then begin error := true ; goto 1 end ;
   .
   .
1:end
```

5.4.6 The With statement

When processing record variables it is quite usual to refer to several fields of the record within a small region of the program. For convenience Pascal provides the **with** statement, within which record field names may be referenced without the record variable name, provided the record variable has been specified at the top of the **with** statement.

with-stat = 'with' record-var { ',' record-var } 'do' statement

Examples :

```
       var person : record
                          name : array [1..10] of char ;
                          age : 1..100 ;
                          sex : (male, female);
                          end ;

       with person do
       begin name := 'Paul Ives '; age := 24; sex := male end
```

After the end of the statement references to fields of the record or records specified in **with** statement must be made in full.

5.5 PROCEDURES AND FUNCTIONS

Pascal, in common with most other programming languages, provides a facility for defining a group of actions in the form of a **procedure**, to which a name, called the procedure identifier, is given. The procedure is activated by a procedure statement, which causes the execution of the group of actions defined in the procedure.

The use of procedures allows us to textually divide the program into sub-units corresponding to the sub-problems identified during construction of the program, thus making the program easier to understand and hence maintain. It also allows us to define once, a piece of code that is executed at different points in the program, thus saving memory space and typing time.

In section 5.2 we said that a block consisted of a declaration part and a statement part, and that within the declaration part was the procedure-and-function-definition-part.

```
      procedure-function-definition-part =
               { procedure-declaration | function-declaration }
```

Functions are dealt with in this section 5.5.2

5.5.1 Procedures

A procedure declaration is defined as follows :

procedure-declaration = procedure-heading (block | 'forward')

The procedure heading consists of the word **procedure**, followed by a name for the procedure and an optional parameter list. Parameters are dealt with later in this section. Following the heading can be the word **forward** or a block. The use of forward declarations is also dealt with later in this section. A block is the same as the program block described earlier and consists of a declaration part and a statement part. Any variables declared in the procedure block are local to the procedure and cannot be accessed outside the procedure. Note that since a declaration part can contain procedure declarations, procedures can be declared inside procedures, the scope of procedure names being the same as the scope of any other variables.

A procedure is actived by a procedure statement, which consists of the procedure name and a list of parameters, if there are any.

Example :

```
procedure readandwritename ;
   const namelen = 30 ;
   var i : integer ;
       ch : char ;

   begin
       for i := 1 to namelen do
       begin
          read ( ch ) ;
          write ( ch )
       end
   end ;
```

To activate the procedure we would use the statement

```
readandwritename ;
```

5.5.1.1 Parameters

Often it is necessary to introduce new variables to represent procedure arguments and results. These variables are called parameters and are defined in the procedure header by specifying a formal parameter list. The formal parameter list specifies the name of each parameter, followed by its type. When a procedure with parameters is called, the procedure statement must contain the procedure name and a list of **actual paramters**. The form of each actual parameter is determined by the class of the corresponding formal parameter. In Computer One Pascal there are two classes of formal parameters – variable parameters and value parameters.

Value parameters can be considered as input parameters, since their only role is to pass values into a procedure. The actual parameter may be any expression which produces a value that is assignment compatible with the corresponding formal parameter. When a procedure is called with a value parameter the formal parameter is assigned the value and while the value of the formal parameter may change during the execution of the procedure, this has no effect on the actual parameter.

Variable formal parameters are used to denote actual parameters whose values may change during the execution of the procedure. Each corresponding actual parameter must therefore be a variable of the same type as the formal parameter. A variable formal parameter is specified in the procedure heading by preceding the parameter name by the symbol **var**. Note that file parameters must always be var parameters.

When specifying the actual parameter list in a procedure call the following rules must be observed :

(1) The number of parameters in the two lists must be the same.

(2) Each actual parameter corresponds to the formal parameter occupying the same position in the formal parameter list.

(3) Corresponding actual parameters must agree as described above for value parameters and for variable parameters.

The examples below shows a procedure with variable and formal parameters and some calls to the procedure.

```
procedure max ( var m : integer ; x, y : integer ) ;
begin
   if x > y then m := x
   else m := y
end ;

The following calls to the procedure max are legal :

max ( size [a], 8, 16 );
max ( maxval, 8*3, 12-6 );
max ( p^.height, a, b ) ; { a and b are integers }
```

```
The following calls to the procedure max are illegal :
       max ( 8, 16,20 ); {actual parameter for a variable
                               formal parameter must be a
                               variable}

       max ( maxval, 8, 'v' ); { actual parameter type does not
                               correspond to formal paramter
                               type }
```

Note that a procedure can call itself, or call another procedure which in turn calls the first procedure. Such a procedure is called a recursive procedure.

5.5.1.2 Forward declarations

If a procedure A calls a procedure B, which it turn calls A either directly or indirectly, this is called mutual recursion. This presents a slight problem in Pascal since the scope rules do not allow a name to be used before it is declared, and with mutually recursive procedures one of the procedures must make a call to procedure which has not yet been declared. To get round this problem Pascal allows a procedure to be declared **forward**, i.e. the procedure header, followed by the symbol **forward** can be placed before the actual procedure declaration itself. In the forward declaration, the formal parameter list, if there is one, must be specified and the formal parameter list must not be specified for the actual procedure declaration. Note that for every forward declaration there must be a corresponding actual declaration of the procedure.

Example :

```
       procedure a ( x,y : integer ) ; forward ;

       procedure b ;
       begin
          .
          .
          a ( 3, 5 )  ; { call to forward declared proc }
          .
          .
       end ;

       procedure a ; {NOTE parameter list must not be repeated}
       begin
          .
          b ;  { call procedure b }
          .
          .
       end ;
```

5.5.2 Functions

A function is a special form of procedure which describes a computation and produces a single value as a result. The type of the result must be a scalar or pointer type. However, whereas a procedure is activated by a procedure statement, a function is activated by a function designator from within an expression, the result of the function contributing to the final value of the expression.

A function declaration is similar to a procedure declaration, but following the formal parameter list the type of the result is specified.

Example :

```
        function max ( x, y : integer ) : integer ;
```

The function identifier is used within the function body to denote the result of the computation. Within the body of the function there must be at least one assignment of a value of the result type to the function identifier. If during execution this assignment is not executed the result of the function is undefined.

Example :

```
      function max ( x, y : integer ) : integer ;
      begin
         if x > y then max := x else max := y
      end ;
```

The function body, as with the procedure body, is a block and may therefore contain declarations of variables, procedures, functions etc. Parameter passing is the same for functions as procedures and the rules stated for procedures also apply to functions.

Note that although the function identifier is assigned to, it cannot be treated as an ordinary variable and any use of the function identifier in an expression will result in a recursive call to the function.

If a function alters the value of a variable which is known outside the function, this is called a side-effect of the function. Care should be taken with side-effects, since if the altered value appears in the same expression as the function designator, the resultant value of the expression will depend on the order of evaluation of the operands. For example if the variable a, which is in scope outside a function f, is altered inside the function f then

f (x) + a   may not give the same value as   a + f (x)

For this reason functions with side-effects should be avoided wherever possible.

5.6 INPUT AND OUTPUT

Every computer program manipulates data and there must be some means of supplying the program with data and receiving results from the program. Data can be input from devices such as microdrive cartridges and keyboards and can be output to devices such as microdrive cartridges and VDUs.

Pascal uses an input stream and an output stream to obtain and deliver information and, to a large extent, the type of device on which the information is held, is immaterial. Any stream of information held on an external device, for input to, or output from a Pascal program, is a file. Note that a keyboard can be considered as a file since it can supply a program with a stream of information.

5.6.1 Input

A Pascal read statement takes the form

```
read ( file, variable-list )
```

For the moment we shall only consider the file **input** which is predeclared in Pascal and is the default input file. In Computer One Pascal the **input** file is from the QL keyboard. For the standard input file a read statement has the form

```
read ( input, variable-list )          or
read ( variable-list )
```

Since the input file is the default file it can be omitted from the read statement. The variable list is defined as

```
variable-list = variable { ',' variable }
```

The variables must be of type integer, real, char or a subrange of type integer or char, and are written as sequences of ASCII characters which conform to the numbers and string constants described in section 5.1.

The effect of the read statement is to assign to each of the variables in the list, values from the input stream. If a variable v is of type integer or real, a sequence of characters representing an integer or real number is read, leading blanks and end of line characters being skipped. All numbers read must be separated by blanks or end of lines. If v is of type char, the next character in the input stream is assigned to v.

If the current character is the end of line marker, the standard function

```
eoln ( input )
```

becomes true. If a character is read when **eoln** is true the character returned is a blank. Note that the values encountered in the input stream must be assignment compatible with the variables in the variable list, otherwise an error will occur.

A special form of the read statement is provided to skip over the remainder of a line. The statement is

```
readln ;
```

Readln can also take a list of variables.

```
read ( v1, v2, v3 ) ;                        is equivalent to
read ( v1 ) ; read ( v2 ) ; read ( v3 ) ;
```

and

```
read ( v1, v2, v3 ) ; readln ;               is equivalent to
readln ( v1, v2, v3 ) ;
```

Example : The statement

```
read ( integer1, integer2, char1, char2, real1, ) ;
```

with the following input

```
36        28XY3.67
```

is equivalent to

```
integer1 := 36 ; integer2 := 28 ; char1 := 'X' ;
char2 := 'Y';
real1 := 3.67 ;
```

The statement will also read the following input correctly.

```
36
28XY
3.6 45
```

The 45 will not be discarded and will be used next time a read statement is executed. However, if the statement had been a **readln** and not a **read**, the 45 would automatically have been skipped over after the other values had been read.

5.6.2 Output

A Pascal write statement takes the form

write (file, output-list)

For the moment we will only consider the predeclared file **output**, which like the input file, is the default and need not be specified in the write statment.

The statement

writeln

may be used to output a new line character and has the more general form

writeln (output-list)

which causes the new line to be output after the output-list. The output-list is a list of output-values, separated by commas. The output-values, which are expressions, must be of type real, integer, string-array, boolean or char.

Examples :

```
write ( 'Forty = ', 8*5 )
```

will output

```
Forty = 40
```

```
writeln ( 'ABC' ); write ( 6=3*2); write ( 'xyz' );
```

will output

```
ABC
truexyz
```

The exact number of characters output for each value is determined by the way the value is expressed.

output-value = expr [':' field-width [':' fraction-part]]

Field-with and fraction-part must be expressions of type integer. The field-width specifies the number of characters spaces used to write out the value. When an output value is written with no field-width a default field-width is used.

For Computer One Pascal the following default widths are used :

type	**default width**
integer	10
real	16
char	1
string	The length of the string array
boolean	4 or 5 (for 'true' or 'false')

If the actual value to be output requires fewer characters than the field-width an appropriate number of blanks is output before the value. If the specified field-width is too small for integer or real values the field width is increased to the minimum required. For a string or boolean the right-most characters are truncated if the field-width is too small.

The fraction-part of the output-value may only be used when outputting real numbers and specifies the number of characters after the decimal point. If no fraction-part is specified the value is output in floating point form. Thus if no fraction-part is specified the output would be of the form

```
3.456E+03
```

If the fraction-part was 2 the value would be output as

```
345.60
```

Remember that there are a number of special characters which help to format output (see 5.1.3).

So far we have only looked at the standard input and output files. We shall look more generally at Pascal files.

5.6.3 Files

As stated in section 5.3.7 a Pascal file can be declared to be of any component type t (except a file iself or a structure with a file field).

```
type f = file of t ;
```

The declaration of a file f automatically introduces a buffer variable f ↑. f ↑ can be considered as a window through which existing components of type t can be read, or new components of type t can be written. The buffer variable component is the only immediately accessible component of the file f.

Pascal files are sequential files, i.e. the file components are either read or written in strict sequential order, and reading and writing to a file cannot be interspersed.

When the window f ↑ has moved beyond the end of a file the standard end-of-file function

eof (f)

is true, otherwise it is false.

The following standard procedures exist for file handling :

attach (f, fname)	–	Computer One Pascal uses this procedure to associate a file with a filename. Fname is a string-array of any length up to forty-one characters which gives the name of the file. The file can be a microdrive file, a console, a screen or any other named device, for example the serial port. Consoles, screens and other character devices should be declared as textfiles (see 5.6.3.1).

Example :
```
        attach ( f, 'MDV2_myfile_dat' )
        attach ( f, 'SCR_512x256a0x0' )
```

reset (f)	–	This sets the buffer variable to the start of the file and, except for console files, assigns the first component to the buffer variable f ↑ . If the file is not empty eof (f) becomes false, otherwise it becomes true and f ↑ is undefined. Attach must have been used to associate this file variable with an existing file.

rewrite (f)	–	This procedure is used before first writing to a file. The current value of f is replaced by the empty file. i.e. the contents of the file are erased. Eof (f) becomes true and f ↑ is undefined. Attach must have been used to associate the file variable with a non-existent file.
get (f)	–	This procedure advances the window to the next component of the file. If no next component exists eof (f) becomes true and f ↑ is undefined.
put (f)	–	This procedure appends to f the value of f ↑ , after which eof (f) remains true and f ↑ is undefined.
close (f)	–	This procedures closes the file, after which attach must be used to associate the file with a name before it is used again.
delete (fname)	–	This procedure deletes the named file.
read (f, var list)	–	Similar to the read statement described in 5.6.1, but the var list must all be variables of the type of the file. Read (f, var) is equivalent to `var:=f^ ; get ( f )`
write (f, list)	–	Similar to the write statement described in 5.6.2, but values in the list must all be of the type of the file. Write (f, val) is equivalent to `f^:=val ; put ( f )`

Example : Suppose we have a file of records, the declaration of the record being

```
type    person = record
                   name : array [1..8] of char;
                   sex : ( male, female ) ;
                   age : 1..100 ;
                 end ;
```

and that someone whose name is K.Jones has been married and we wish to change the name in the record to K.Smith. Since Pascal does not allow reading and writing to the same file, we have to read the records from one file, change the name of the required record and write the records to a new file.

```
program changename ;

       type  person = record
               name : array [1..8] of char;
               sex : ( male, female ) ;
               age : 1..100 ;
             end ;

       var   pfile1, pfile2 : file of person;
             per : person;

       begin
          attach ( pfile1, 'mdv2_persons' ) ;
          reset ( pfile1 ) ;    { prepare the file for reading }
          attach ( pfile2, 'mdv2_persons2' ) ;
          rewrite ( pfile2 ) ;  { prepare the file for writing}
          while not eof ( pfile1 ) do
          begin
             per := pfile1^ ;
             get ( pfile1 ) ;
             with per do
             if (name = 'K.Jones ') and (sex = female) then
               name := 'K.Smith ';
             pfile2^ := per ;
             put ( pfile2 )
          end ;
          close ( pfile1 ) ; close ( pfile2 ) ;
       end.
```

Note that we could have used

read (pfile1,per) and write (pfile2,per)

instead of assignments using the file buffer variable and the **get** and **put** statements.

5.6.3.1 Text files

A text file is a file whose components are characters. In Pascal there is a predeclared type defined as

```
type text = file of char
```

Text files are usually divided into lines separated by control characters. In Computer One Pascal the line separator character is the ASCII control code 10 (line feed). The standard **input** and **output** files described earlier are predeclared textfile identifiers.

Like other files, textfiles must be prepared for writing using the standard procedures **attach** and **reset** or **rewrite**. Note that **reset** and **rewrite** must not be used for the standard **input** and **output** files.

The standard procedures **eof**, **get**, **put** and **close** may also be applied to textfiles. Likewise a textfile f has an associated buffer variable f ↑ . Note however that when using **reset** from a character device f, such as the keyboard, f ↑ is not assigned the first component of f, since the buffer variable cannot have the next input component until it has been entered at the keyboard.

For a character device, such as a keyboard, eof (f) is always false.

The function eoln (f) returns true when the next character available is the end of line control character, or if the textfile is a console when the current character is the end of line character. If eoln (f) is true the character blank (= chr (32)) is assigned to the variable of the read statement.

The standard procedures **readln** and **writeln** can only be used for textfiles. As with the standard textfiles, **input** and **output**, **read** and **readln** can only be used to read variables of type integer, real or char and the output procedures **write** and **writeln** can only be used to write variables of type integer, char, boolean, string and real, if the file is a text file. There is a standard procedure page which outputs a new page character (chr (12)) to a textfile. This procedure is useful when the file is a print file.

Computer One has a number of predeclared procedures and functions which allow you to make full use of the windowing and other I/O facilities on the QL. These procedures and functions are described in chapter 6.

5.6.3.2 Standard Input and Output channels

The standard I/O channnels **input** and **output** are set up to use the same file 'con_', which is a console with parameters '448x180a32x16'. Since this window does not use the whole screen another window, for example a debugging window, can be used on another part of the screen without disturbing the contents of the default I/O window. NOTE that the **input** and **output** files may be redefined using the **attach** and either **reset** or **rewrite** procedures, but the current **input** and **output** channels should not be closed before doing so.

5.6.4 I/O Errors

Computer One Pascal detects errors which occur in I/O operations. However, it is the user's responsibility to check for this. There is a predeclared integer, **ioresult**, which contains the error code after an input or output operation has been performed. **Ioresult** can be inspected by the user program. A value of 0 indicates that the operation completed successfully, otherwise the error code indicates what the error was. A list of the errors and their causes is given in Appendix D. Note that when checking for an error condition the following must not be used :

```
        if ioresult <> 0 then write ( 'ioresult = ', ioresult )
```

since the operation to write the error code will corrupt the value. Instead, **ioresult** should be assigned to another variable and that variable should be used in the write statement.

CHAPTER SIX

STANDARD PROCEDURES FOR THE QL

This chapter contains a description of the predeclared procedures and functions which have been added to Computer One Pascal to enable you to make full use of the power of the QL from Pascal.

The following types have been predeclared and are used as parameter types for some of the procedures and functions to be described :

```
type    address = 0..1048575 ;  { = 2^20 - 1 }
        colour = ( black, blue, red, magenta, green,
                          cyan, yellow, white ) ;
        dreg = array [0..7] of integer ;
        areg = array [0..5] of integer ;
```

6.1 QL I/O AND GRAPHICS PROCEDURES AND FUNCTIONS

The following procedures correspond to the procedures available in SuperBasic. Note, however, that all the parameters of the functions must be specified. Most of the procedures take a textfile which has been opened as a console or a screen. Further information on these procedures can be found by looking up the procedure name in the KEYWORDS section of the QL user-manual. All of the procedures which take a file parameter set the predeclared **ioresult** variable described in 5.6.4.

For any procedure which takes a file name parameter, a string array of any length up to 41 characters may be used. Characters up to the first blank or line-feed in the string are used. If the procedure operates on a window the operation takes place in the window attached to the specified file.

In the following three procedures the value of 'stipple' is in the range 0..3. For values outside the range only the bottom two bits of the given integer are used.

procedure paper (var f : file ; maincol, contrastcol : colour ; stipple : integer) ;

Sets a new paper and strip colour for the given file. The paper colour is used by the cls procedure and will remain in effect until the next use of paper with this file.

procedure ink (var f : file ; maincol, contrastcol : colour ; stipple : integer) ;

Sets a new ink colour for the given file. This is the colour in which any text output to the window will appear.

procedure strip (var f : file ; maincol, contrastcol : colour ; stipple : integer) ;

Sets a new strip colour for the given file. This is the background colour for any text written to the window . It is rather like highlighting the text.

procedure window (var f : file ; width, depth, x, y : integer) ;

Allows the user to change the size and position of the window. The four coordinates are specified in pixels and the position is relative to the screen origin.

procedure border (var f : file; width : integer ; maincol,
contrastcol : colour ; stipple : integer);

Adds a border, in the given colour, to the window. For subsequent operations on the window, except another border operation, the window size is reduced to allow space for the border.

procedure cls (var f : file; part : integer) ;

Clears the specified part of the window. The part cleared will be set to the colour set with last call to paper for this window.

where if part = 0 the whole window is cleared,
if part = 1 the part above the cursor line is cleared,
if part = 2 the part below the cursor line is cleared,
otherwise the whole window is cleared.

procedure at (var f : file; lineno, columno : integer) ;

Sets the print position in the window. The next text written to the window after the call will start at the specified position. Position 0,0 is the top left corner of the window.

procedure scroll (var f: file; part: integer; numpixels: integer);

This procedure scrolls the window up or down the specified number of pixels,

where if part = 0 the whole window is scrolled,
if part = 1 the part above the cursor line is scrolled,
if part = 2 the part below the cursor line is scrolled,
otherwise the whole window is scrolled

procedure pan (var f: file; part: integer; numpixels :integer);

Pans the window the specified number of pixels,

where if part = 0 the whole screen is panned,
if part = 3 the whole cursor line is panned,
if part = 4 the right end of the cursor, including the cursor position is panned,
otherwise the whole screen is panned.

procedure under (var f : file; on : boolean) ;

Turns underlining on in the window, if the **on** parameter is true. If the **on** parameter is false underlining is switched off.

procedure over (var f : file; switch : integer) ;

Selects the type of printing required and remains in effect until **over** is next called,

where if switch = 0 characters are printed with the strip colour
as background,
if switch = 1 characters are printed with the paper colour
as background,
if switch = −1 characters are printed over the previous
contents of the window,
otherwise switch is assumed to be zero.

procedure flash (var f : file; on : boolean) ;

Turns character flashing on in the window, if the **on** parameter is true. If the **on** parameter is false flashing is switched off. **Flash** is only effective in low resolution mode.

procedure csize (var f : file; width, height : integer) ;

Sets the new character size for the window.

width param	**width in pixels**	**height param**	**height in pixels**
0	6	0	10
1	8	1	20
2	12		
3	16		

procedure recol (var f: file; c0,c1,c2,c3,c4,c5,c6,c7: colour);

Recolours all the pixels in the window,

where c0 is the new colour for all black pixels.
c1 is the new colour for all blue pixels
c2 is the new colour for all red pixels
c3 is the new colour for all magenta pixels
c4 is the new colour for all green pixels
c5 is the new colour for all cyan pixels
c6 is the new colour for all yellow pixels
c7 is the new colour for all white pixels

The following nine procedures are graphics procedures. The graphics origin of a window is (0,0) and is the bottom left corner of the window. The origin can be altered using the **scale** procedure. All positions are relative to the graphics origin and not to the current graphics plot position.

procedure arc (var f : file ; sx, sy, fx, fy, angle : real) ;

Draws the arc of a circle in the window from the start point (sx,sy) to the finish point (fx,fy). The 'angle' parameter is the angle subtended by the arc.

procedure block (var f : file ; width, height, x, y : integer ;
main, contrast : colour ; stipple : integer) ;

Fills a block of the specified width and height at the specified (x,y) position in the window

procedure gcursor (var f : file ; x, y, xrel, yrel : real) ;

Positions the screen cursor at the position (xrel,yrel) relative to the graphics position (x,y). Note that xrel and yrel are specified in the pixel coordinate system and x and y in the graphics coordinate system.

procedure cursor (var f : file ; x, y : real);

Positions the screen cursor at the specified (x,y) position, x and y being specified in the pixel coordinate system.

procedure ellipse (var f: file; x, y, radius, eccentricity, angle : real) ;

Draws an ellipse at the specified (x,y) position in the window. If the 'eccentricity' parameter is 1 a circle is drawn. (See under keyword 'Circle' in the QL User Manual).

procedure grafill (var f : file; fill : boolean) ;

Switches the 'graphics fill' on or off. If the value is true shapes are filled when drawn. (See under keyword 'Fill' in QL User Manual).

procedure line (var f : file ; sx, sy, fx, fy : real) ;

Draws a line from the specified start point (sx,sy) to the finish point (fx,fy) in the window.

procedure point (var f : file ; x, y : real) ;

Draws a point at the specified position (x,y) in the window.

procedure scale (var f : file ; scalev, x, y : real) ;

Alters the scale factor of the window. The default scale is 100. i.e. the window is divided vertically into 100 units. The position (x,y) specifies the graphics origin.

procedure mode (resolution : integer) ;

Sets the resolution of the screen and the number of colours that can be displayed,

where if resolution = 8 or 256 resolution is low,
if resolution = 4 or 512 resolution is high,
otherwise resolution is low

procedure baud (rate : integer) ;

Sets the baud rate for communication via the two serial channels. Both channels are set to the same baud rate. Allowable values for the baud rate are

75, 300, 600, 1200, 2400, 4800, 9600.

The baud rate 19200 can be used for transmitting only. The effect is undefined for other values.

function inkey (var f : file; duration : integer) : char ;

Waits for the given duration for a character from the given file f. The character, if input from the keyboard, will not be echoed on the screen. The duration is specified in fiftieths of a second.

6.2 MEMORY ACCESS PROCEDURES AND FUNCTIONS

The following procedures allow access to memory, with the exception of the function **byte**, and should be used with extreme caution, since ANY area of memory can be accessed.

function byte (whichbyte: integer; from : integer) : integer

Takes a values in the range 0..3 and selects that byte value from the given integer. Byte 0 is the most significant byte of the given integer.

procedure call (add : address; var datareg : dreg ; var addrreg : areg);

Calls a machine code subroutine, whose start address is the specified address. The **datareg** and **addrreg** arrays contain the values which will be loaded into the registers d0-d7 and a0-a5. The types **dreg** and **areg** are predeclared array types. Calling this procedure causes the registers to be loaded from the arrays and a 'JSR' instruction to be executed. The machine code routine should return using an 'RTS' instruction. The stack pointer value (a7) immediately before the RTS must be the same value as on entry. The stack beneath this must NOT be changed in any way. On return, the arrays **datareg** and **addrreg** contain the values which were in d0-d7 and a0-a5 when the machine code subroutine finished.

procedure fill (count : integer; from : address; ch: char) ;

Fills **count** bytes of memory with the character **ch** starting at the specified address.

procedure move (count : integer ; from, tto : address) ;

Moves **count** bytes from the address **from** to the address **tto** ;

function peek (add : address) : integer
function peekw (add : address) : integer
function peekl (add : address) : integer

The three functions return the byte, the word and longword at the specified address. Addresses are rounded down to an even boundary for **peekw** and **peekl**.

procedure poke (add : address ; val : integer)
procedure pokew (add : address ; val : integer)
procedure pokel (add : address ; val : integer)

The three procedures poke the given value, a byte, word or long word into memory at the given address. Addresses are rounded down to an even boundary for **pokew** and **pokel**. For **pokew** and **pokeb** the least significant word and byte of the given value are poked into memory.

function loc (variable-name) : address ;

Returns the address of the given variable. The variable may be of any type.

function getbytes (var f: file; count: integer; addr: address) : integer ;

Attempts to get the specified number of bytes from the given file and puts it in memory starting at the given address. Returns the number of bytes actually read. This function sets the **ioresult** variable.

function getline (var f : file; count: integer; addr: address) : integer ;

Reads the specified number of bytes from the given file, or a sequence of bytes terminated by a line-feed character. Returns the number of bytes read, including the line-feed. **ioresult** is set to 'buffer-full' (−5) if no line-feed is read. This function allows the cursor control keys to edit the line if it is entered from the keyboard.

procedure putbytes (var f: file; count: integer; addr: address);

Copies to the given file, the specified number of bytes, starting at the given address.

procedure sbytes (name : filename; start : address ; length : integer) ;

Saves to the given file the number of bytes specified by length and starting at the given address. If the file already exists it will be overwritten.

procedure lbytes (name : filename ; start address) ;

Loads the specified file into memory at the given address.

6.3 OTHER QL PROCEDURES AND FUNCTIONS

procedure beep (pitch1, pitch2, interval, duration, step, wrap,
rand, fuzzy : integer) ;

Activates the QL sound functions. The best way to to use this procedure is to experiment with different parameter values. The range of values for each of the parameters is as follows :

duration :	−32768..32767
pitch1 :	0..255
pitch2 :	0..255
wrap :	0..15
interval :	0..15
step :	−8..7
fuzzy :	0..15
rand :	−32768..32767

procedure bell ;

Causes the QL to emit a short beep.

function isTV : boolean ;

Returns true if the TV option was chosen when the QL was reset.

function islowres : boolean ;

Returns true if the TV or monitor being used is currently in low resolution mode.

function digit (ch : char) : boolean ;

Returns true if the given char is a digit, i.e. if the char is in the range '1'..'9'. It returns false otherwise.

function letter (ch : char) : boolean ;

Returns true if the given char is a letter, lower or upper case, false otherwise.

function lower (ch : char) : char ;

Returns the given character in lower case. If the character is not a letter the function returns the given character.

function upper (ch : char) : char ;

Returns the given character in upper case. If the character is not a letter the function returns the given character.

procedure randomise (seed : integer) ;

Provides a seed for the random number generator.

function rnd (r : integer) : integer

Returns a random number between 0 and r−1.

function rad (degrees : real) : real

Converts the given angle in degrees to radians.

function deg (radians : real) : real

Converts the given angle in radians to degrees.

procedure halt

Halts execution of the program.

procedure setclock (secs : integer) ;

Sets the clock to the given number of seconds.

procedure adjustclock (secs : integer) ;

Adds the given number of seconds to the clock.

function readclock : integer ;

Reads the clock.

CHAPTER SEVEN

STANDARD PASCAL PROCEDURES AND FUNCTIONS

This chapter contains a description of the standard Pascal procedures and functions to be found in this and most Pascal implementations.

7.1 FUNCTIONS

Fn name	Argument type	Action	type of result
abs	integer or real	Absolute value of arg.	integer or real
arctan	integer or real	Arctangent of arg.	real
chr	integer	character whose ascii code is the arg.	char
cos	integer or real	Cosine of arg.	real
eof	file	true if end of file reached	boolean
exp	integer or real	e to power arg.	real
ln	integer or real	natural log of arg.	real

Fn name	Argument type	Action	type of result
odd	integer	true if arg. is odd	boolean
ord	ordinal type	ordinal value of arg.	integer
pred	ordinal type	predecessor of arg.	same ordinal type
round	real	round to nearest integer	integer
sin	integer or real	sine of arg.	real
sqr	integer or real	square of arg.	integer or real
sqrt	integer or real	square root of arg.	real
succ	ordinal type	successor of arg.	same ordinal type
trunc	real	truncated value of arg.	integer

7.2 PROCEDURES

Proc name	Argument type(s)	Action
get	file	assigns the next component of the file to the buffer variable.
new	pointer and optional tags	Allocates storage on the heap for a dynamic variable of the type pointed at by the pointer.
page	textfile	Writes a new page character (chr (12)) to the given file.

Proc name	**Argument type(s)**	**Action**
put	file	Appends the contents of the variable buffer of the file to the file.
read	file and variable list	Reads data items from the file into the variables in the list.
readln	textfile and variable list	Similar to read, but moves to next end of line indicator.
dispose	pointer	releases the space allocated on the heap for the dynamic variable pointer at by the given pointer.
reset	file	positions file buffer to start of file.
rewrite	file	creates the file, and prepares it to accept output.
write	file and value list	writes data from the list to the file.
writeln	textfile and value list	writes data from the list to the file and terminates it with an end of line.

BIBLIOGRAPHY

Pascal from BASIC	P.J Brown (Addison-Wesley)
Introduction to Pascal	Welsh & Elder (Prentice-Hall)
Pascal User Manual and Report	Jensen & Wirth (Springer-Verlag)
Programming in Pascal	P.Grogono (Addison-Wesley)
Software Tools in Pascal	Kernigan & Plauger (Addison-Wesley)
Specification for the Computer programming Language Pascal	British Standards Institution BS 6192:1982

APPENDIX A
COMPILER ERROR MESSAGES

This appendix lists all the compiler error numbers and the corresponding messages.

1:	error in simple type
2:	identifier expected
3:	'program' expected
4:	')' expected
5:	':' expected
6:	illegal symbol
7:	error in parameter list
8:	'of' expected
9:	'(' expected
10:	error in type
11:	'[' expected
12:	']' expected
13:	'end' expected
14:	';' expected
15:	integer expected
16:	'=' expected
17:	'begin' expected
18:	error in declaration part
19:	error in field list
20:	'.' expected
21:	'*' expected
50:	error in constant
51:	':=' expected
52:	'then' expected
53:	'until' expected
54:	'do' expected
55:	'to'/'downto' expected
58:	error in factor
59:	error in variable

101:	identifier declared twice
102:	low bound exceeds high bound
103:	identifier not of appropriate class
104:	identifier not declared
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 type
113:	index type must be scalar or subrange
114:	base type must not be real
115:	base type must be scalar or subrange
116:	error in type of standard procedure parameter
117:	unsatisfied forward reference
119:	forward declared; repetition of parameter list not allowed
120:	function result type must be scalar, subrange or pointer
121:	file value parameter not allowed
122:	forward declared function; repetition of result type not allowed
123:	missing result type in function declaration
124:	F-format for real only
125:	error in type of standard function parameter
126:	number of parameters does not agree with declaration
129:	type conflict of operands
130:	expression is not of set type
131:	tests on equality allowed only
132:	strict inclusion not allowed
133:	file comparison not allowed
134:	illegal type of operand(s)
135:	type of operand must be Boolean
136:	set element type must be scalar or subrange
137:	set element types not compatible
138:	type of variable is not array
139:	index type is not compatible with declaration
140:	type of variable is not record
141:	type of variable must be file or pointer
142:	illegal parameter substitution
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 not be integer
150:	assignment to standard function is not allowed
152:	no such field in this record
153:	type error in read
154:	actual parameter must be variable
155:	control variable must not be declared on intermediate level
156:	multidefined case label
157:	too many cases in case statement
158:	missing corresponding variant declaration
159:	real or string tagfields not allowed
160:	previous declaration was not forward
161:	again forward declared
162:	parameter size must be constant
165:	multidefined label
166:	multideclared label
167:	undeclared label
168:	undefined label
169:	error in base set
171:	standard file was redeclared
177:	assignment to function identifier not allowed here
178:	multidefined record variant
180:	control variable must not be formal
201:	error in real constant: digit expected
202:	string constant must not exceed source line
203:	integer constant exceeds range
205:	empty string not allowed
206:	integer part of real constant exceeds range
207:	label too long
250:	too many nested scopes of identifiers
251:	too many nested procedures and/or functions
255:	too many errors on this source line
261:	further errors surpressed
262:	too many subranges
304:	element expression out of range
398:	implementation restriction
399:	not implemented

APPENDIX B

RUN-TIME ERRORS

This appendix gives all the run-time error messages and their possible causes.

1. **Out of Store**

   – this error is output when their is no space left for dynamic variables on the heap or there are too many nested procedure calls, causing the stack to run out of space.

2. **UnknownInstruction**

   – An unimplemented standard procedure or function function has been used.

3. **Arithmetic Overflow**

   – An integer expression has a value greater than maxint or less than -maxint

   or a real expression has a value outside the range of reals.

4. **Divide by zero**

   – An attempt has been by to divide by zero.

5. **Bad Number**

   – A read statement, expecting an integer or a real value received an invalid value.

Example : If x is an integer the statement

```
read ( x )
```

will cause this error if the input is 'x'.

6. **Value outside range**

– An attempt has been made to index an array with array with a value outside the array bounds

or an attempt has been made to a assign a value to a variable which is outside the range of values allowed for that variable.

or the value of the selector expression of a case statement was less than the minimum value or greater than the maximum value of the case statement labels.

Example :

```
      var      a : array [1..10] of integer ;
               x : 1..20 ;
               .
               .
               a [2*6] := x ;  { 12 outside index range-
 1..10 }
               x := 12 - 12 ;  { 0 outside range 1..20 }
```

7. **Case value outside range**

– The value of the selector expression of a case statement was within the minimum and maximum value of the case labels, but was not equal to any of the labels.

Example :

```
      ch := '5' ;
      case ch of
               '1', '2','3' : ........ ;
               '7', '8','9' : ........ ;
      end ;
      .
      .
```

This produces the error because '5' lies between '1' and '9', but is not one of the values of the case statement labels.

8. **Nil pointer** – An attempt has been made to access a dynamic variable using a pointer with a value **nil**.

Example :

```
        type person = record
                        sex : ( male, female ) ;
                        age : 1..100 ;
                        end ;
        var p = ^person ;
                    .
                    .
        p := nil ;
        p^.age := 20 ; { this causes the error since p is nil }
```

The standard procedure new should be used to allocate space for the dynamic variable and to point p to the variable.

9. **End of file** – An attempt has been made to read from a file when eof (f) was already true. i.e. an attempt to read past the end of the file.

10. **Odd Address** – The start address in lbytes or sbytes call is odd

11. **Can't open default channel**
    – When a program starts to run there is insufficient memory to open the standard input and output files. Try rebooting.

12. **File not open for reading**
    – Read operation attempted on file not open for reading.

13. **File not open for writing**
    – Write operation attempted on file not open for writing.

14. **File not attached**
    – Attempt to open a file which has not been attached.

15. **File already open**
    – Attempt to open a file that is already open.

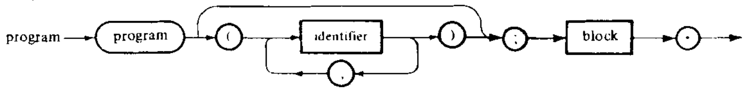
APPENDIX C

COMPUTER ONE PASCAL SYNTAX DIAGRAMS

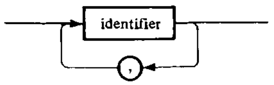
Syntax diagrams provide a convenient notation for defining the syntax of a programming language. The rule for using a syntax diagram is very simple – if you can make up a set of symbols by following the arrows of a syntax diagram then that set of symbols is syntactically correct.

In the diagrams names that are in rectangular boxes refer to other syntax diagrams ; symbols or words in circles or ovals are the basic symbols of the language. The name on left hand side of the diagram says what syntactic entity the diagram is defining.

Example ;

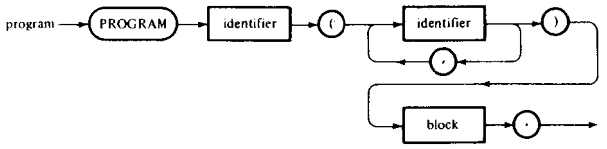


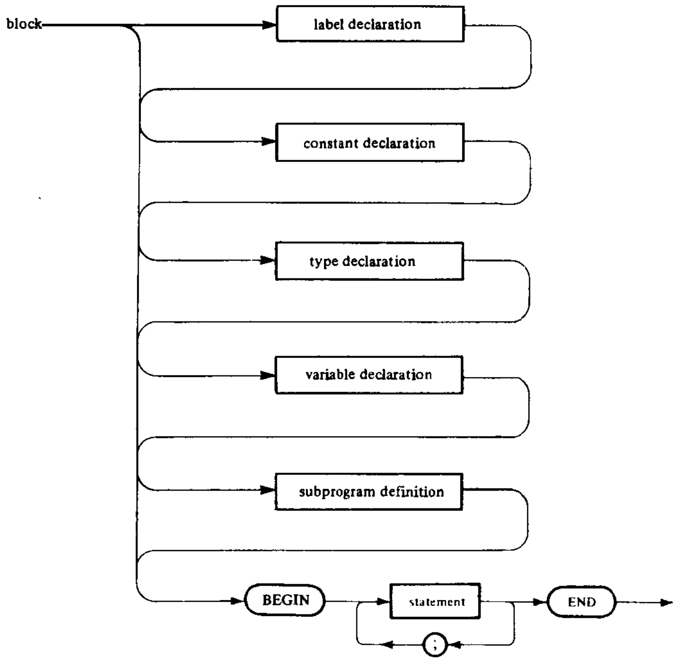
From this diagram it can be seen that a program consists of the reserved word **program**, followed optionally by a list of identifiers, separated by commas and enclosed in brackets, followed by a semi-colon, a block and a period. The arrow going round the bracketed identifier list indicates that the identifier list need not be specified. The part of the diagram describing the identifier list

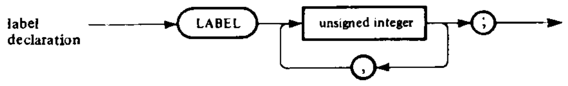


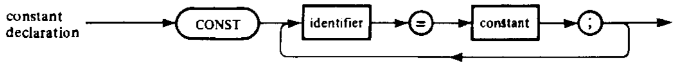
indicates that there can be any number of identifiers separated by commas.

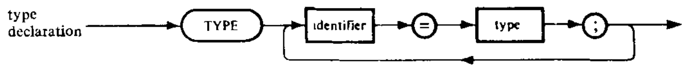
The full syntax of Computer One Pascal is given on the following pages.

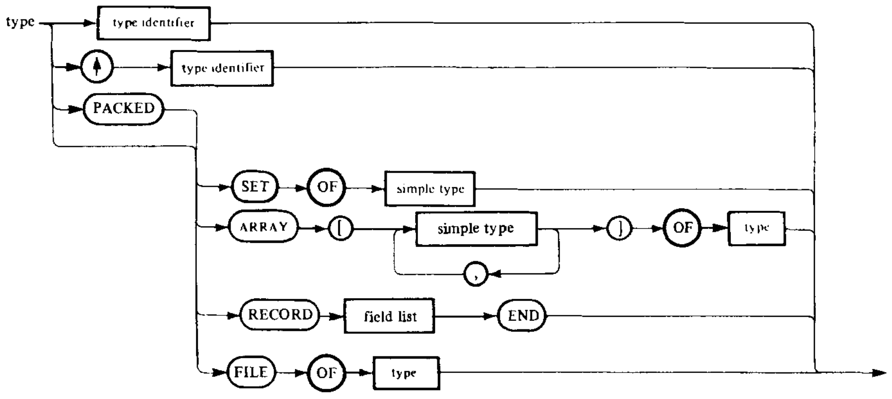

program
PROGRAM
identifier
(
identifier
,
)
block
.

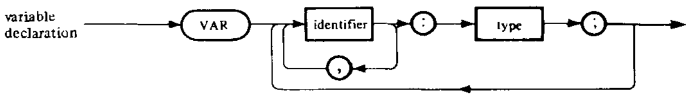

block
label declaration
constant declaration
type declaration
variable declaration
subprogram definition
BEGIN
statement
;
END

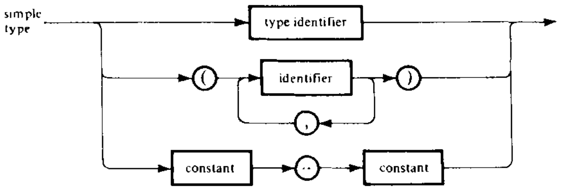

label declaration
LABEL
unsigned integer
,
;

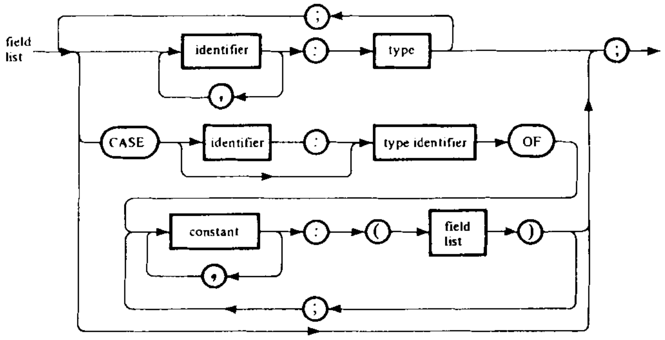

constant declaration
CONST
identifier
=
constant
;

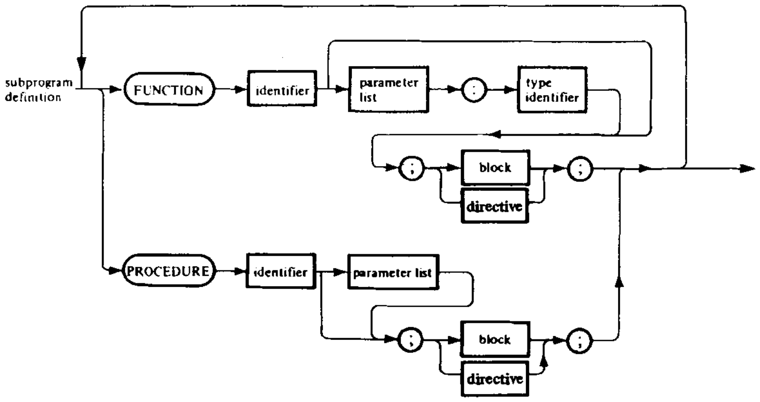

type declaration
TYPE
identifier
=
type
;

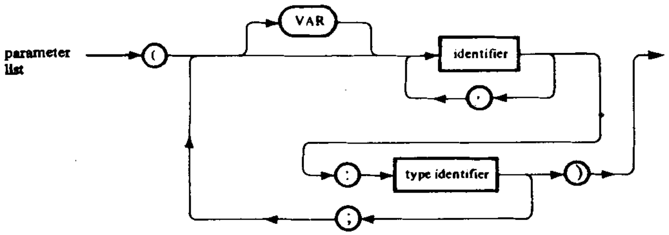

type
type identifier
↑
type identifier
PACKED
SET
OF
simple type
ARRAY
[
simple type
,
]
OF
type
RECORD
field list
END
FILE
OF
type

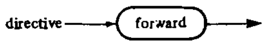

variable declaration
VAR
identifier
,
:
type
;

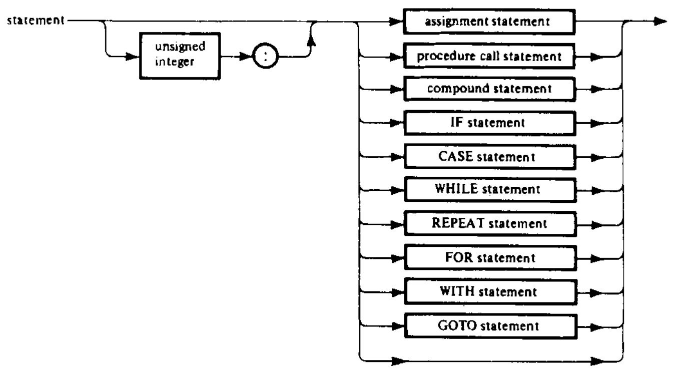

simple type
type identifier
(
identifier
,
)
constant
..
constant

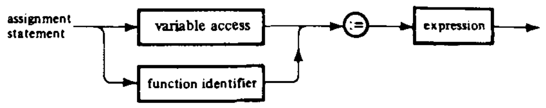

field list
identifier
,
:
type
;
;
CASE
identifier
:
type identifier
OF
constant
,
:
(
field list
)
;

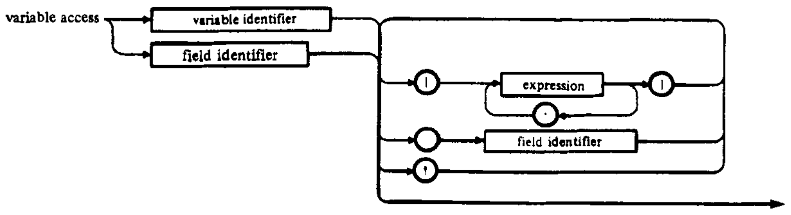

subprogram definition
FUNCTION
identifier
parameter list
:
type identifier
;
block
directive
;
PROCEDURE
identifier
parameter list
;
block
directive
;

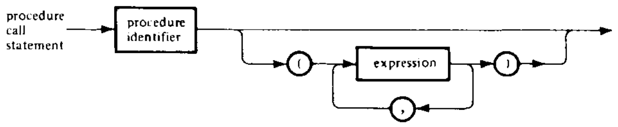

parameter list
(
VAR
identifier
,
:
type identifier
)
;

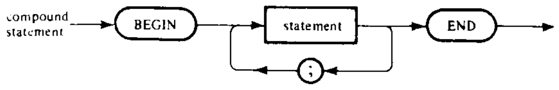

directive
forward

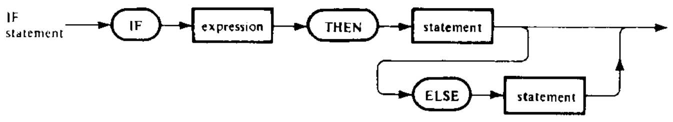

statement
unsigned integer
:
assignment statement
procedure call statement
compound statement
IF statement
CASE statement
WHILE statement
REPEAT statement
FOR statement
WITH statement
GOTO statement

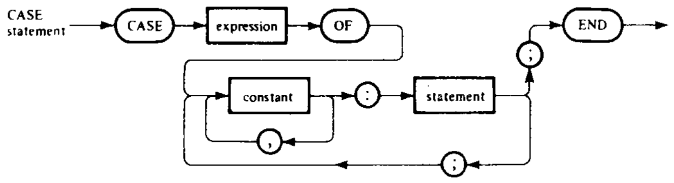

assignment statement
variable access
function identifier
:=
expression

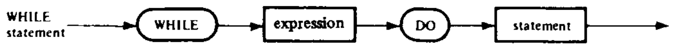

variable access
variable identifier
field identifier
[
expression
,
]
.
field identifier
↑

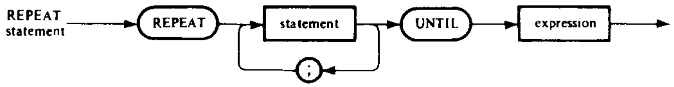

procedure call statement
procedure identifier
(
expression
,
)

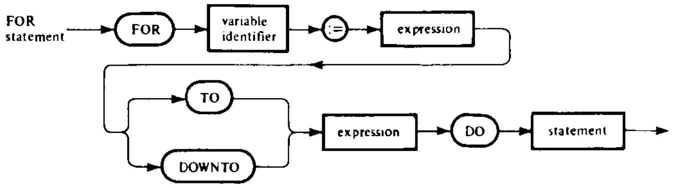

compound statement
BEGIN
statement
;
END

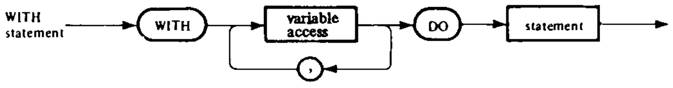

IF statement
IF
expression
THEN
statement
ELSE
statement

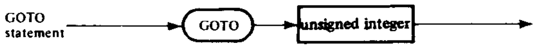

CASE statement
CASE
expression
OF
constant
,
:
statement
;
;
END

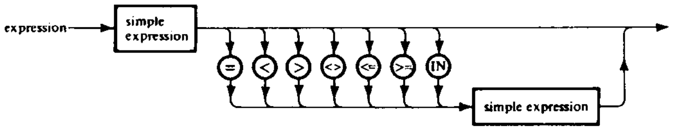

WHILE statement
WHILE
expression
DO
statement

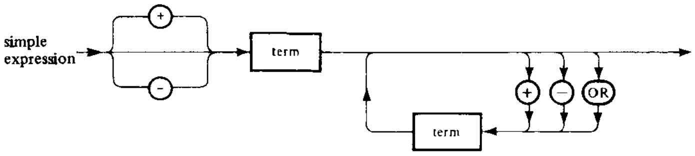

REPEAT statement
REPEAT
statement
;
UNTIL
expression

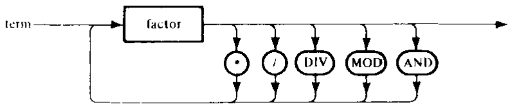

FOR statement
FOR
variable identifier
:=
expression
TO
DOWNTO
expression
DO
statement

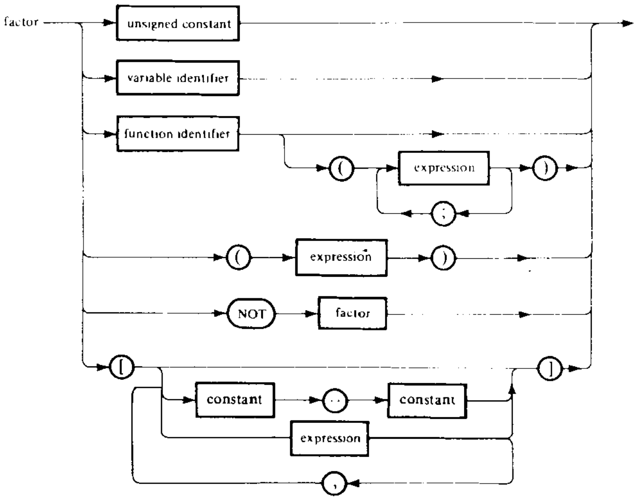

WITH statement
WITH
variable access
,
DO
statement

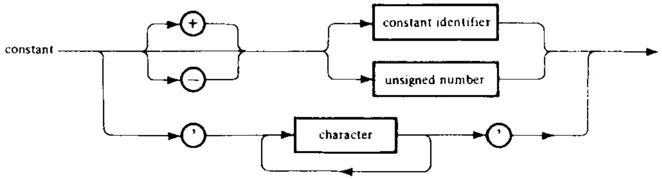

GOTO statement
GOTO
unsigned integer

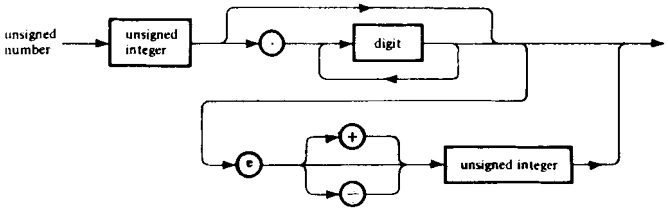

expression
simple expression
=
<
>
<>
<=
>=
IN
simple expression

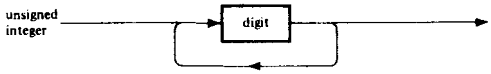

simple expression
+
-
term
+
-
OR
term

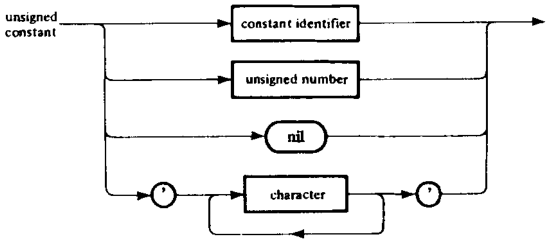

term
factor
*
/
DIV
MOD
AND

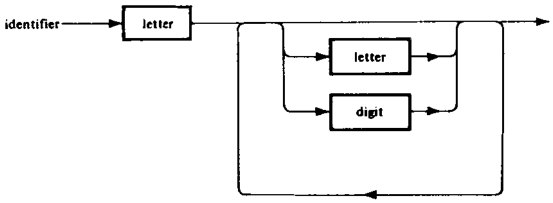

factor
unsigned constant
variable identifier
function identifier
(
expression
;
)
(
expression
)
NOT
factor
[
constant
..
constant
expression
,
]


constant
+
-
constant identifier
unsigned number
'
character
'


unsigned number
unsigned integer
.
digit
e
+
−
unsigned integer


unsigned integer
digit


unsigned constant
constant identifier
unsigned number
nil
'
character
'


identifier
letter
letter
digit

APPENDIX D

SYSTEM ERRORS

This appendix gives the QDOS system error messages. The error number given for each message is the value to which the predeclared variable **ioresult** is set when an error occurs during an I/O operation. Note that the error code 1 is not a QDOS error.

Number	Error Message	Possible Cause
1		An invalid reset or rewrite was attempted on a file. For example an attempt to reset a screen, which is a write only device.
−1	not complete	An I/O operation did not complete before it timed out.
−2	invalid Job	
−3	out of memory	The QL has no memory available.
−4	out of range	Parameters for creating a window were invalid values.
−5	buffer overflow	The input is longer than the input buffer.
−6	channel not open	Specified channel has not been opened.
−7	not found	File does not exist or no cartridge in microdrive.
−8	already exists	Attempt to create a file that already exists.
−9	in use	The specified file is already being used.
−10	end of file	The last component of a file has already been read.
−11	drive full	Microdrive cartridge is full.
−12	bad name	Specified device does not exist.

Number	Error Message	Possible Cause
−13	transmission error	
−14	format failed	
−15	bad parameter	
−16	bad or changed medium	
−17	error in expression	
−18	arithmetic overflow	
−19	not implemented	
−20	read only	Cartridge is write protected.

APPENDIX E
EXAMPLE PROCEDURES

This appendix gives a number of procedures which shows how some of the QL specific features of Computer One Pascal are used.

```
program openfile ;
const  maxlen = 41 ;
var    f : text ;
       fname : array [1..maxlen] of char ;
       errcode : integer ;

procedure getfilename ;
{ prompts for a file name and attempts to open it }
var    inlength : integer ;
begin
   write ( 'Input file name : ' ) ;
   { use getline to get the name so that edit line features can
     be used }
   inlength := getline ( input, maxlen, loc (fname) ) ;
   attach ( f, fname ) ;
   rewrite ( f ) ; { open for writing }
end ;

begin
   repeat { keep trying to open file until success }
       getfilename ;
       if ioresult <> 0 then
       begin
          errcode := ioresult ;
          write ( 'Open file failure : errcode =',
          errcode:3 ) ;
       end

       else errcode := 0 ;
   until errcode = 0 ;
   { the file f is now open for writing. Assume it is a
     window }
   { now clear the window, set up the ink and paper colour and
       draw an ellipse }
   paper ( f, white, white, 0 ) ; { no stipple }
   ink ( f, black, black, 0 ) ;
   cls ( f, 0 ) ; { clear the whole window }

   { draw ellipse at ( 50,50 ) with radius 30, eccentricity 0.5
       and angle 45 degrees }
   ellipse ( f, 50, 50, 30, 0.5, 45 ) ;
   close ( f ) ;
end.
```

```
program machinecodecall ;
const  codelen = 512 ; { machine code routine no more than-
 1/2K }
var    addrregs : areg ; { address registers }
       dataregs : dreg ; { data registers }
       code : array [1..codelen] of char ;
begin
   { load the machine code routine into the code array }
   lbytes ( 'mdv2_mcroutine_cde', loc ( code ) ) ;
   { if necessary could set up register values here }
   call ( loc ( code ), dataregs, addrregs ) ;
   { can now examine registers after the routine has returned }
end.
```

INDEX

Computer One – Software Problem Report – PASCAL

Name	Return to :
Address	Computer One Ltd.,
	Science Park,
....	Milton Road,
	Cambridge CB4 4BH.
....	

Telephone Number :

Nature of Problem (tick): Documentation error[ ] Software error[ ]

QDOS Version No. _____

Master Cartridge Name..

Software Error : Please describe problem in as much detail as possible, giving the keystroke sequence which caused the error. (enclose listing if possible) –

Documentation Error : Please include page number in error description

Comments or Enquiries :