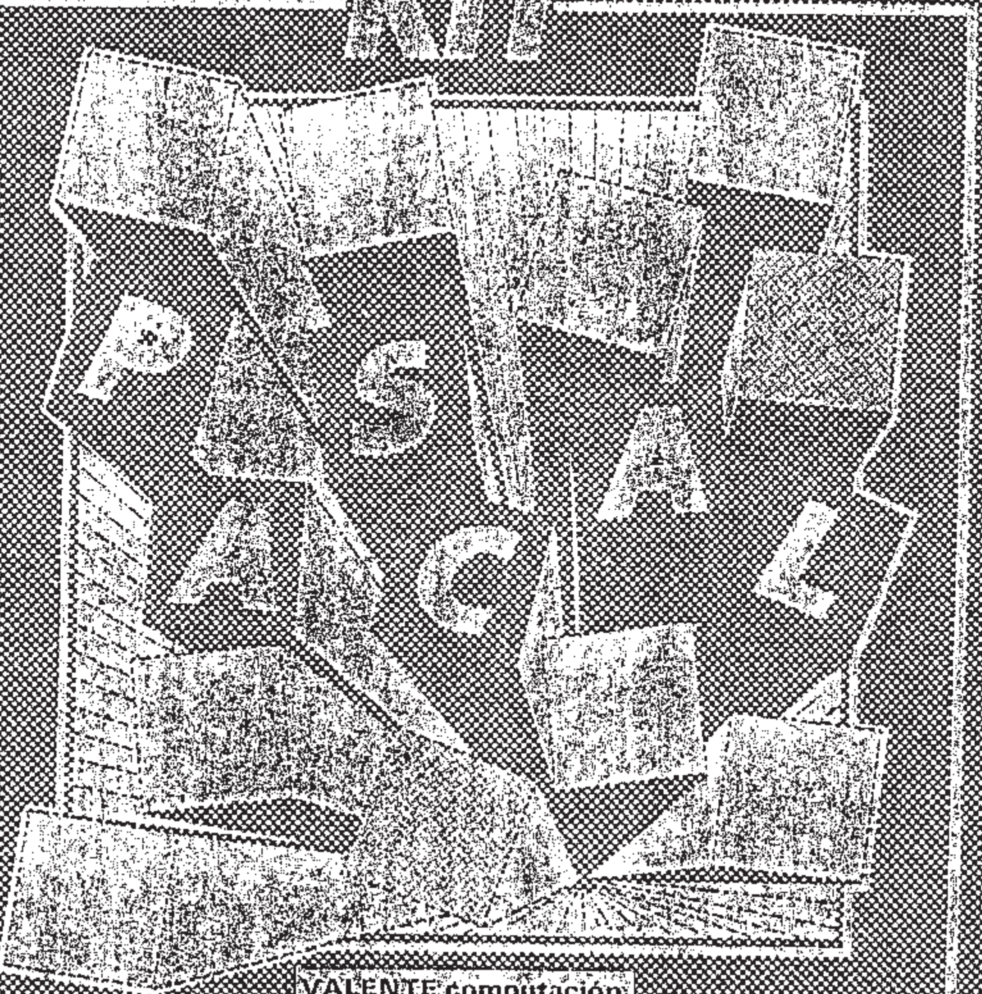
QL PASCAL DEVELOPMENT KIT



VALENTE computación
Caleruega, 8 - 28033 MADRID
Telef. 202 67 01

METACOMCO

QL PASCAL DEVELOPMENT KIT

Contents

Using QL Pascal

Welcome to QL Pascal. In your development kit you should have the following items:

1. Microdrive cartridge A: Pascal compiler
2. Microdrive cartridge B: Screen Editor and Pascal run-time system
3. An EPROM containing part of the QL Pascal
4. The *QL PASCAL Development Kit* manual

We strongly recommend that you make backup copies of the two microdrives cartridges and keep the master copies in a safe place.

Use of the EPROM cartridge

As has been mentioned above, part of the Pascal is provided on an EPROM. The QL Pascal EPROM is encased in a plastic cartridge which can be inserted into the machine whenever the language is required.

To insert your EPROM, first **POWER DOWN** your QL. This is very important! Then remove the cover from the QL socket marked "ROM" which is located on the left at the rear of the computer. The EPROM cartridge can now be pushed carefully into this socket. Once the cartridge is in, the machine can be powered up.

Having selected the required screen (either TV or monitor) the EPROM can now be verified. It is not essential to do this. However, as part of it may be missing or damaged, it is a good idea to check. To verify your EPROM, type:

```
ROM
```

The EPROM will then run a check on itself. If it is working, the message "QL PASCAL VERSION", followed by the version number, will appear on the top left hand corner of the screen. If the EPROM is faulty then the message "BAD ROM" will appear and you should contact Metacomco for further assistance. After the EPROM has been verified, the language can be used.

Each EPROM is internally numbered according to its version. The same numbering system is used for the compiler. When a program is compiled, the compiler checks to see if the EPROM version number tallies with its own. It also checks the installation. If either the number fails to tally, or the installation is in any way incorrect, the compiler will return an appropriate error.

The Pascal Compiler

Preparing to run the Pascal compiler

The microdrive cartridge containing the compiler (A) should be inserted into the left-hand drive and the cartridge containing your Pascal program should be inserted into the right-hand drive. Note that it is possible to change the default drive on which the compiler resides (see the section on the Install program).

Running the compiler

The QL Pascal compiler is invoked by specifying.

```
exec_w <drive_no>_pascal
```

or

```
exec <drive_no>_pascal
```

Following the initial loading of the compiler, CTRL-C must be entered to position the cursor at the first compiler prompt (unless exec__w has been used). This asks for the name of the input source file, which must be specified in accordance with QDOS file name syntax.

The compiler on receiving this source file name checks for __PAS as the final extension and if it is absent adds it. It then attempts to open <filename>__PAS. If it fails and it added __PAS it tries to open <filename> as the source file.

The following five prompts are then generated in turn:

- i) Listing file?
- ii) Code file?
- iii) Omit range-checking [Y/N]?
- iv) Extensions to ISO standard [Y/N]?
- v) Workspace size?

The enter key may be depressed in response to these prompts, if the default conditions are required.

i) Listing file?

If an output compilation listing file is required then the file name for the listing must be specified. The listing file defaults to <filename>__LST if the supplied filename does not end in __LST. The default is no listing file.

ii) Code file?

The output code file name, if required, is specified here. The default is no code file. Thus the compiler can be used for syntax checking only. The code file defaults to <filename>__REL if the supplied filename does not end in __REL.

iii) Omit range-checking [Y/N]?

Programs generally execute more efficiently if range-checking is turned off but are prone to unpredictable results if data value mismatches are encountered. The default response is 'N'.

iv) Extensions to ISO standard [Y/N]?

Type 'Y' if the use of the ISO standard extensions is required. The default response is 'N'.

v) Workspace size?

Enter an integer to specify the workspace size in bytes, or an integer followed by 'K' to specify the workspace required in kilobytes. The default size for a QL with 128K of memory has been set to 20K. For large programs we recommend that you use memory expansion on your QL.

Compilation

During compilation on an unexpanded (128K) QL you will notice the compiler using the screen area of memory as workspace. This will not usually happen on a QL with memory expansion.

Any errors, apart from warnings, detected by the compiler cause suppression of further code generation. All error numbers and the erroneous portion of source text are displayed in a thin window on the console together with the prompt:

```
Press ENTER (continue) or A (abort)
```

If a listing file is specified at compile time the error number is output at the appropriate point in the compilation listing. A table of error messages corresponding to error numbers can be found in Appendix B.

At the end of compilation the following prompt appears:

```
[illegible] files to compile [Y/N]?
```

If more files are to be compiled then type 'Y' otherwise type 'N'. The default is 'N'. CTRL C must now be entered to reposition the cursor at the main QL command line.

The list file gives useful information about the compilation. It supplies;

i) the name of the file compiled.

ii) a listing of the source code with each line, statement and level of logical nesting numbered.

iii) any compilation errors found, positioned at the relevant place in the source listing.

iv) details of the block structure of the program, procedures, functions and associated storage.

v) details of the identifers declared (in the main program, procedures and functions) and associated storage.

Running a program

Linking in the Pascal run-time library

Before object code is ready for execution, the Pascal run-time library must be linked in using the QL Pascal linker. To do this the microdrive cartridge containing the program PASLINK (Cartridge B) should be inserted into one of the drives.

To run PASLINK, type:

```
exec_w <drive no>_paslink
```

or

```
exec <drive no>_paslink
```

Once loaded, PASLINK will request the name of a binary file. This should be the output from a previous run of the compiler. On receiving the binary file name, the compiler checks for _REL as the final extension and if it is absent, adds it. It then attempts to open <filename>_REL. If it fails and it added _REL, it tries to open <filename> as the binary file. Once satisfied on the first input name it will ask for a further binary file input. In the simple case of a single segment program you should now press ENTER. You will then be asked for an output file name, which is where the linked program will be placed. This output will contain your program and the complete Pascal runtime system, and will be a code file which is directly runnable using EXEC or EXEC_W.

If you want to make a code file then enter a suitable file name. You will next be asked for the stack size to be used. The stack is used for all main program variables and the default is 800 longwords.

If you are still developing a program, the next step after creating a code file with PASLINK would be to run it, and so PASLINK allows a shortcut here. If you simply press ENTER in response to the request for the output file, PASLINK will load your program and then execute it immediately. Before it starts your program the window will be cleared, and you can start testing. There is a restriction on this use, which is that the stack space used will not be alterable and that more space than is needed will be taken up because both PASLINK and your program are in store simultaneously.

If you wish to include external procedures written in Metacomco BCPL or Assembler, then you should enter the filename of the Pascal object code file as the first input file, and then instead of immediately pressing ENTER when asked for a further input file you should provide the next file name and so on. A reponse of just ENTER terminates the list.

Program execution

At run-time the QL Pascal run-time library is loaded. If a run-time error is detected program execution is terminated and an error message is displayed on the console.

Changing the default window

The editor allows the window to be altered as part of the initialisation sequence. If this option is not required then the default window is used. This is initially the same as the window used during the start of the program, but if required the default window may be altered permanently by patching the programs. This is useful where a certain window size and position is always required and means that the window does not have to be positioned correctly each time the program is run. The default windows used by PASCAL and PASLINK can also be patched.

Changing the default drive names

For those users who upgrade their QLs with disc drives, there is the possibility of changing default drive names to something other than mdv1. This option will not be given when installing the editor ED since it can be EXECed from any device.

If the default device is changed for PASCAL then the compiler will look on the new device for its overlays. If the default device is changed for PASLINK then if your Pascal program creates any temporary files, they will appear on the new device. The default device may be changed to somethimg other than mdv1 by patching the relevant program.

The INSTALL program

The program INSTALL is supplied on the distribution microdrive cartridge (B) to perform both of the above tasks. It is run by the command

```
LRUN <drive no>_install
```

The program starts by asking whether the default window is to be set up for TV or monitor mode. The minimum window size is greater in TV mode because the characters used are larger. You should answer T if you are setting the default for use with TV mode and M if you are setting it for use with monitor mode. Note that the current mode in use is of no consequence.

The standard window will appear on the screen and can be moved by means of the cursor keys and altered in size by means of ALT cursor keys. Once the window is in the right place and of the desired size, press ENTER.

The program now asks for the name of the file which is to be modified. If you wished to alter the editor then the file would probably be something like mdv1_ed. The next item requested is the name of the program. When a new job such as the editor or Pascal is running on the QL it has a name associated with it. This can be inspected by suitable utilities. The name is six characters long and whatever is typed here is used as the name and forced to the correct length. The name is of little importance except for job identification.

[illegible] PASCAL and PASLINK the program will then go on to [illegible] drive name. If you do not wish to change the default [illegible] reply should be

```
MDV1_
```

[illegible] the reply must be (MDV1_). If you do wish to change the default drive name the reply should be the device name, for example

```
FLP1_
```

[illegible] PASCAL this will append FLP1_ to its overlays before attempting to load them.

The INSTALL program will then modify the file specified. INSTALL can be run as many times as you like to alter the default window. It is unlikely to be useful with programs other than those distributed by Metacomco that provide user selection of an initial window such as Metacomco's Assembler, LISP, BCPL and Pascal.

Chapter 1: The Screen Editor

1.1 Introduction

The screen editor ED may be used to create a new file or to alter an existing one. The text is displayed on the screen, and can be scrolled vertically or horizontally as required. The size of the program is about 20K bytes and it requires a minimum workspace of 8K bytes.

The editor is invoked using EXEC or EXEC_W as follows

```
EXEC_W mdv1_ed
```

The difference between invoking a program with EXEC or EXEC_W is as follows. Using EXEC_W means that the editor is loaded and SuperBasic waits until the editing is complete. Anything typed while the editor is running is directed to the editor. When the editor finishes, keyboard input is directed at SuperBasic once more.

Using EXEC is slightly more complicated but is more flexible. In this case the editor is loaded into memory and is started, but SuperBasic carries on running. Anything typed at the keyboard is directed to SuperBasic unless the current window is changed. This is performed by typing CTRL-C, which switches to another window. If just one copy of ED is running then CTRL-C will switch to the editor window, and characters typed at the keyboard will be directed to the editor. A subsequent CTRL-C switches back to SuperBasic. When the editor is terminated a CTRL-C will be needed to switch back to SuperBasic once more. More than one version of the editor can be run concurrently (subject to available memory) if EXEC is used. In this case CTRL-C switches between SuperBasic and the two versions of the editor in turn.

Once the program is loaded it will ask for a filename which should conform to the standard QDOS filename syntax. No check is made on the name used, but if it is invalid a message will be issued when an attempt is made to write the file out, and a different file name may be specified then if required. All subsequent questions have defaults which are obtained by just pressing ENTER.

The next question asks for the workspace required. ED works by loading the file to be edited into memory and sufficient workspace is needed to hold all the file plus a small overhead. The default is 12K bytes which is sufficient for small files. The amount can be specified as a number or in units of 1024 bytes if the number is terminated by the character K. If you ask for more memory than is available then the question is asked again. The minimum is 8K bytes.

You are next asked if you wish to alter the window used by ED. The default window is normally the same as the window used in the initialisation of ED although this may be altered if required. If you type N or just press ENTER then the default window is used. If you type Y then you are given a chance to alter the window. The current window is displayed on the screen and the cursor keys can be used to move the window around. The combination ALT and the cursor keys will alter the size of the window although there is a minimum size which may be used. Within this constraint you can specify a window anywhere on the screen so that you can edit a file and do something else such as run a SuperBasic program concurrently. When you are satisfied with the position of the window press ENTER.

Next an attempt is made to open the file specified, and if this [illegible] then the file is read into storage and the first few lines [illegible] on the screen. Otherwise a blank screen is provided, ready for [illegible] of new data. The message File too big indicates that more [illegible] should be specified.

[illegible] the bottom line of the screen is used as a [illegible] command line. Any error messages are displayed [illegible] until another editor command is given.

[illegible] immediate commands and [illegible] commands. Immediate commands are those which are executed [illegible] by a single key or control key [illegible] commands are typed in onto the command line [illegible] the command line is finished. A number of [illegible] on a single command line, and any [illegible] may be grouped together and groups repeated automatically. [illegible] immediate commands have a matching extended version.

Immediate commands use the function keys and cursor keys on the QL in conjunction with the special keys SHIFT, CTRL and ALT. For example, delete line is requested by holding down the CTRL and ALT keys and then pressing the left arrow key. This is described in this document as CTRL-ALT-LEFT. Function keys are described as F1, F2 etc.

The editor attempts to keep the screen up to date, but if a further command is entered while it is attempting to redraw the display, the command is executed at once and the display will be updated later, when there is time. The current line is always displayed first, and is always up to date.

1.2 Immediate commands

Cursor control

The cursor is moved one position in either direction by the cursor control keys LEFT, RIGHT, UP an DOWN. If the cursor is on the edge of the screen the text is scrolled to make the rest of the text visible. Vertical scroll is carried out a line at a time, while horizontal scroll is carried out ten characters at a time. The cursor cannot be moved off the top or bottom of the file, or off the left hand edge of the text.

The ALT-RIGHT combination will take the cursor to the right hand edge of the current line, while ALT-LEFT moves it to the left hand edge of the line. The text will be scrolled horizontally if required. In a similar fashion SHIFT-UP places the cursor at the start of the first line on the screen, and SHIFT-DOWN places it at the end of the last line on the screen.

The combinations SHIFT-RIGHT and SHIFT-LEFT take the cursor to the start of the next word or to the space following the previous word respectively. The text will be scrolled vertically or horizontally as required. The TAB key can also be used. If the cursor position is beyond the end of the current line then TAB moves the cursor to the next tab position, which is a multiple of the tab setting (initially 3). If the cursor is over some text then sufficient spaces are inserted to align the cursor with the next tab position, with any characters to the right of the cursor being shuffled to the right.

Inserting text

Any letter typed will be added to the text in the position indicated by the cursor, unless the line is too long (there is a maximum of 255 characters in a line). Any characters to the right of the text will be shuffled up to make room. If the line exceeds the size of the screen the end of the line will disappear and will be redisplayed when the text is scrolled horizontally. If the cursor has been placed beyond the end of the line, for example by means of the TAB or cursor control keys, then spaces are inserted between the end of the line and any inserted character. Although the QL keyboard generates a different code for SHIFT-SPACE and SHIFT-ENTER these are mapped to normal space and ENTER characters for convenience.

An ENTER key causes the current line to be split at the position indicated by the cursor, and a new line generated. If the cursor is at the end of a line the effect is simply to create a new, empty blank line after the current one. Alternatively CTRL-DOWN may be used to generate a blank line after the current, with no split of the current line taking place. In either case the cursor is placed on the new line at the position indicated by the left margin (initially column one).

A right margin may be set up so that ENTERs are automatically inserted before the preceding word when the length of the line being typed exceeds that margin. In detail, if a character is typed and the cursor is at the end of the line and at the right margin position then an automatic newline is generated. Unless the character typed was a space, the half completed word at the end of the line is moved down to the newly generated line. Initially there is a right margin set up at the right hand edge of the window used by ED. The right margin may be disabled by means of the EX command (see later).

Deleting text

The CTRL-LEFT key combination deletes the character to the left of the cursor and moves the cursor left one position. If the cursor is at the start of a line then the newline between the current line and the previous is deleted (unless you are on the very first line). The text will be scrolled if required. CTRL-RIGHT deletes the character at the current cursor position without moving the cursor. As with all deletes, characters remaining on the line are shuffled down, and text which was invisible beyond the right hand edge of the screen may now become visible.

The combination SHIFT-CTRL-RIGHT may be used to delete a word or a number of spaces. The action of this depends on the character at the cursor. If this character is a space then all spaces up to the next non-space character on the line are deleted. Otherwise characters are deleted from the cursor, and text shuffled left, until a space is found. The CTRL-ALT-RIGHT command deletes all characters from the cursor to the end of the line. The CTRL-ALT-LEFT command deletes the entire current line.

Scrolling

Besides the vertical scroll of one line obtained by moving the cursor to the edge of the screen, the text may be scrolled 12 lines vertically by means of the commands ALT-UP and ALT-DOWN. ALT-UP moves to previous lines, moving the text window up. ALT-DOWN moves the text window down moving to lines further on in the file. The F4 key rewrites the entire screen, which is useful if the screen is altered by another program besides the editor. Remember that you can switch out of the editor window and into some other job by typing CTRL-C at any point, assuming that there is another job with an outstanding input request. SuperBasic will be available only if you entered the editor using EXEC rather than EXEC_W. If there is enough room in memory you can run two versions of ED at the same time if you wish.

Repeating commands

The editor remembers any extended command line typed, and this set of extended commands may be executed again at any time by simply pressing F2. Thus a search command could be set up as the extended command, and executed in the normal way. If the first occurrence found was not the one required, typing F2 will cause the search to be executed again. As most immediate commands have an extended version, complex sets of editing commands can be set up and executed many times. Note that if the extended command line contains repetition counts then the relevant commands in that group will be executed many times each time the F2 key is pressed.

1.3 Extended commands

Extended command mode is entered by pressing the F3 key. Subsequent input will appear on the command line at the bottom of the screen. Mistakes may be corrected by means of CTRL-LEFT and CTRL-RIGHT in the normal way, while LEFT and RIGHT move the cursor over the command line. The command line is terminated by pressing ENTER. After the extended command has been executed the editor reverts to immediate mode. Note that many extended commands can be given on a single command line, but the maximum length of the command line is 256 characters. An empty command line is allowed; so typing ENTER after F3 will return to immediate mode.

Extended commands consist of one or two letters, with upper and lower case regarded as the same. Multiple commands on the same line are separated from each other by a semicolon. Commands [illegible] followed by an argument, such as a number or a string. [illegible] and terminated by a delimiter [illegible] any character besides letters, numbers, space, semicolon or [illegible]

[illegible] corresponding extended version [illegible] section [illegible]

Program control

The command X causes the editor to exit. The text held in storage is written out to file, and the editor then terminates. The editor may fail to write the file out either because the filename specified when editing started was invalid, or because the microdrive becomes full. In either case the editor remains running, and a new destination should be specified by means of the SA command described below. Alternatively the Q command terminates immediately without writing the buffer; confirmation is requested in this case if any changes have been made to the file. A further command allows a 'snapshot' copy of the file to be made without coming out of ED. This is the SA command. SA saves the text to a named file or, in the absence of a named file, to the current file. For example:

```
     *SA /mdv2_savedtext/
```

or

```
     *SA
```

This command is particularly useful in areas subject to power failure or surge. It should be noted that SA followed by Q is equivalent to the X command. Any alterations made between the SA and the Q will cause ED to request confirmation again; if no alterations have been made the program will be quitted immediately with the file saved in that state. SA is also useful because it allows the user to specify a filename other than the current one. It is therefore possible to make copies at different stages and place them in different files.

The SA command is also useful in conjunction with the R command. Typing R followed by a filename causes the editor to be re-entered editing the new file. The old file will be lost when this happens, so confirmation is requested (as with the Q command) if any changes to the current file have been made. The normal action is therefore to save the current file with SA, and then start editing a new file with R. This saves having to load the editor into memory again, and means that once the editor is loaded the microdrive containing it can be replaced by another.

The U command "undoes" any alterations made to the current line if possible. When the cursor is moved from one line to another, the editor takes a copy of the new line before making any changes to it. The U command causes the copy to be restored. However the old copy is discarded and a new one made in a number of circumstances. These are when the cursor is moved off the current line, or when scrolling in a horizontal or vertical direction is performed, or when any extended command which alters the current line is used. Thus U will not "undo" a delete line or insert line command, because the cursor has been moved off the current line.

The SH command shows the current state of the editor. Information such as the value of tab stops, current margins, block marks and the name of the file being edited is displayed. Tabs are initially set at every three columns; this can be changed by the command ST, followed by a number n, which sets tabs at every n columns. The left margin and right margin can be set by SL and SR commands, again followed by a number indicating the column position. The left margin should not be set beyond the width of the screen. The EX command may be used to extend margins; once this command is given no account will be taken of the right margin on the current line. Once the cursor is moved off the current line, margins are enabled once more.

Block control

A block of text can be identified by means of the BS (block start) and BE (block end) commands. The cursor should be moved to the first line required in a block, and the BS command given. The cursor can then be moved to the last line wanted in the block by cursor control commands or in any other way such as searching. The BE command is then used to mark the end of the block. Note, however, that if any change is made to the text the block start and block end become undefined once more. The start of the block must be on the same line, or a line previous to, the line which marks the end of the block. A block always contains all of the line(s) within it.

Once a block has been identified, a copy of it may be moved into another part of the file by means of the IB (insert block) command. The previously identified block is replicated immediately after the current line. Alternatively a block may be deleted by means of the DB command, after which the block start and end values are undefined. It is not possible to insert a block within itself.

Block marks may also be used to remember a place in a file. The SB (show block) command resets the screen window on the file so that the first line in the block is at the top of the screen.

A block may also be written to a file by means of the WB command. The command is followed by a string which represents a file name. The file is created, possibly destroying the previous contents, and the buffer written to it. A file may be inserted by the IF command. The filename given as the argument string is read into storage immediately following the current line.

Movement

The command T moves the screen to the top of the file, so that the first line in the file is the first line on the screen. The B command moves the screen to the bottom of the file, so that the last line in the file is the bottom line on the screen if possible.

The commands N and P move the cursor to the start of the next line and previous line respectively. The commands CL and CR move the cursor one place to the left or one place to the right, while CE places the cursor at the end of the current line, and CS places it at the start.

It is common for programs such as compilers and assemblers to give line numbers to indicate where an error has been detected. For this reason the command M is provided, which is followed by a number representing the line number which is to be located. The cursor will be placed on the line number in question. Thus M1 is the same as the T command. If the line number specified is too large the cursor will be placed at the end of the file.

Searching and Exchanging

Alternatively the screen window may be moved to a particular context. The command F is followed by a string which represents the text to be located. The search starts at one place beyond the current cursor position and continues forwards through the file. If found, the cursor is placed at the start of the located string. To search backwards through the text use the command BF (backwards find) in the same way as F. BF will find the last occurrence of the string before the current cursor position. To find the earliest occurence use T followed by F; to find the last, use B followed by BF. The string after F and BF can be omitted; in this case the string specified in the last F, BF or E command is used. Thus

```
    *F /wombat/

    *BF
```

will search for 'wombat' in a forwards direction and then in a reverse direction.

The E (exchange) command takes a string followed by further text and a further delimiter character, and causes the first string to be exchanged to the last. So for example

```
    E /wombat/zebra/
```

would cause the letters 'wombat' to be changed to 'zebra'. The editor will start searching for the first string at the current cursor position, and continues through the file. After the exchange is done the cursor is moved to after the exchanged text. An empty string is allowed as the search string, specified by two delimiters with nothing between them. In this case the second string is inserted at the current cursor position. No account is taken of margin settings while exchanging text.

A variant on the E command is the EQ command. This queries the user whether the exchange should take place before it happens. If the response is N then the cursor is moved past the search string. If the response is Y or ENTER then the change takes place; any other response (except F2) will cause the command to be abandoned. This command is normally only useful in repeated groups; a response such as Q can be used to exit from an infinite repetition.

All of these commands normally perform the search making a distinction between upper and lower case. The command UC may be given which causes all subsequent searches to be made with cases equated. Once this command has been given then the search string "wombat" will match "Wombat", "WOMBAT", "WoMbAt" and so on. The distinction can be enabled again by the command LC.

Altering text

The E command cannot be used to insert a newline into the text, but the I and A commands may be used instead. The I command is followed by a string which is inserted as a complete line before the current line. The A command is also followed by a string, which is inserted after the current line. It is possible to add control characters into a file in this way.

The S command splits the current line at the cursor position, and acts just as though an ENTER had been typed in immediate mode. The J command joins the next line onto the end of the current one.

The D command deletes the current line in the same way as the CTRL-ALT-LEFT command in immediate mode, while the DC command deletes the character at the cursor in the same way as CTRL-RIGHT.

Repeating commands

Any command may be repeated by preceding it with a number. For example,

```
    4 E /slithy/brillig/
```

will change the next four occurrences of 'slithy' to 'brillig'. The screen is verified after each command. The RP (repeat) command can be used to repeat a command until an error is reported, such as reaching the end of the file. For example,

```
    RP E /slithy/brillig/
```

will change all occurrences of 'slithy' to 'brillig'.

Commands may be grouped together with brackets and these command groups executed repeatedly. Command groups may contain further nested command groups. For example,

```
RP (F /bandersnatch/; 3 (IB; N))
```

will insert three copies of the current block whenever the string 'bandersnatch' is located.

Note that some commands are possible, but silly. For example,

```
RP SR 60
```

will set the right margin to 60 ad infinitum. However, any sequence of extended commands, and particularly repeated ones, can be interrupted by typing any character while they are taking place. Command sequences are also abandoned if an error occurs.

1.4 Command list

In the extended command list, /s/ indicates a string, /s/t/ indicates two exchange strings and n indicates a number.

Immediate commands

F2	Repeat last extended command
F3	Enter extended mode
F4	Redraw screen
LEFT	Move cursor left
SHIFT-LEFT	Move cursor to previous word
ALT-LEFT	Move cursor to start of line
CTRL-LEFT	Delete left one character
CTRL-ALT-LEFT	Delete line
RIGHT	Move cursor right
SHIFT-RIGHT	Move cursor to start of next word
ALT-RIGHT	Move cursor to end of line
CTRL-RIGHT	Delete right one character
CTRL-ALT-RIGHT	Delete to end of line
SHIFT-CTRL-RIGHT	Delete word to right
UP	Move cursor up
SHIFT-UP	Cursor to top of screen
ALT-UP	Scroll up
DOWN	Move cursor down
SHIFT-DOWN	Cursor to bottom of screen
ALT-DOWN	Scroll down
CTRL-DOWN	Insert blank line

Extended Commands

A/s/	Insert line after current
B	Move to bottom of file
BE	Block end at cursor
BF	Backwards find
BS	Block start at cursor
CE	Move cursor to end of line
CL	Move cursor one position left
CR	Move cursor one position right
CS	Move cursor to start of line
D	Delete current line
DB	Delete block
DC	Delete character at cursor
E/s/t/	Exchange s into t
EQ/s/t/	Exchange but query first
EX	Extend right margin
F/s/	Find string s
I/s/	Insert line before current
IB	Insert copy of block
IF/s/	Insert file s
J	Join current line with next
LC	Distinguish between upper and lower case in searches
M n	Move to line n
N	Move cursor to start of next line
P	Move cursor to start of previous line
Q	Quit without saving text
R/s/	Re-enter editor with file s
RP	Repeat until error
S	Split line at cursor
SA/s/	Save text to file s
SB	Show block on screen
SH	Show information
SL n	Set left margin n
SR n	Set right margin n
ST n	Set tab distance n
T	Move to top of file
U	Undo changes on current line
UC	Equate U/C and l/c in searches
WB/s/	Write block to file s
X	Exit, writing text back

Chapter 2: Introduction to QL PASCAL

2.1 Introduction

Since the publication, some nine years ago, of the `Pascal User Manual and Report` by Kathleen Jensen & Niklaus Wirth, Pascal has achieved widespread application in both educational institutions and the commercial world. Its block-structured nature together with the stability and efficiency of its implementation provide for a suitable systematic medium required by teaching establishments; the resulting program readability and maintainability satisfy the long-term high-level development requirement of the commercial software world. The degree of interest shown by commerce indicates the extent to which Wirth succeeded in his aim to produce a simple overall language concept. The User Manual and Report became the unofficial standard Pascal definition which, in certain areas, was open to interpretation by implementors.

Following snowballing interest in Pascal by commercial developers, accompanied by growing concern over portability between the increasing number of different implementations available, the British Standards Institute sponsored the drafting of a Pascal standard. This was eventually published in 1982 as the ISO standard specification which in clarifying the 'grey' areas in Wirth's definition provides a complete, precise and accepted definition of Pascal.

QL PASCAL is an implementation of standard Pascal prepared in accordance with the International Standards Organization standard ISO 7185 / BS 6192. Enhancements have been included in order to produce a convenient environment for the development of structured, efficient and maintainable software to which the use of the Pascal Language lends itself.

The compiler is single pass and produces full MC68000 native code. Integers are 32 bits wide. Sets can comprise of up to a quarter of a million elements. The 24-bit addressing capability of the MC68000 can be utilised to, for example, manipulate large RAM resident arrays. Software developed using QL PASCAL, enhancements apart, is easily transportable, at the source code level, to other implementations of Pascal conforming to the ISO standard but the substantial processing environment afforded by the MC68000 microprocessor must be borne in mind when contemplating such an exercise for a different target processor and associated operating system.

This manual fully describes the QL PASCAL language and language related topics independently of the implementation and operating environment. It has been organized with speed and ease of reference in mind and therefore is not intended to act as a Pascal tutorial guide if you are new to computer programming; however it may be used as such if you have experience of high-level language programming.

Two excellent textbooks on Pascal are:

Brown P J (1982) Pascal from Basic
Addison-Wesley, London, ISBN 0-201-13789-5

Cooper D (1983) Standard Pascal User Reference Manual
W W Norton & Co, London, ISBN 0-393-30121-4

Chapter 3: Language Guide

3.1 Language overview

This section combines a broad description of the syntactic components of the language together with a description of its block-structured nature to provide a logical overview for the purposes of referencing the detailed language description that follows in later sections.

Notational conventions

With reference to syntax descriptions, the following notational conventions are used throughout this manual:

UPPER CASE	Words in upper case are QL PASCAL 68000 reserved words or predefined identifiers
lower case	Variable information is in lower case
n	A numeric value. The default is decimal
[illegible]	Precise literal information is enclosed by, but does not include, double quotes
{}	Items inside curly brackets are optional and can occur zero or more times. Items not contained within curly or square brackets are mandatory.
[]	Items inside square brackets are optional but cannot occur more than once. Items not contained within square or curly brackets are mandatory.
<>	Items inside angle brackets represent variable syntactic constructs detailed elsewhere in the manual.

[illegible] One item is required from the choice of items delimited by round brackets.

[illegible] The vertical bar signifies a choice between the items it separates.

[illegible] Horizontal dots signify continuation.

[illegible] Vertical dots signify continuation.

Tokens

The smallest individual units or tokens of QL Pascal 68000 consist of the following three basic types:

i) Special symbols

These are:

[illegible]

ii) Word symbols or reserved words

These are:

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

iii) Identifiers

These may be of any length and all of the characters used are significant. They must start with an alphabetic character and can continue with a mixed collection of alphabetic characters or digits. Blanks and special symbols cannot be included. Alphabetic characters are the upper and lower case letters of the English alphabet. Digits are 0 - 9.

Reserved words and identifiers can be specified using upper or lower case characters or a mixture of both. Upper case characters are not distinct from lower case characters.

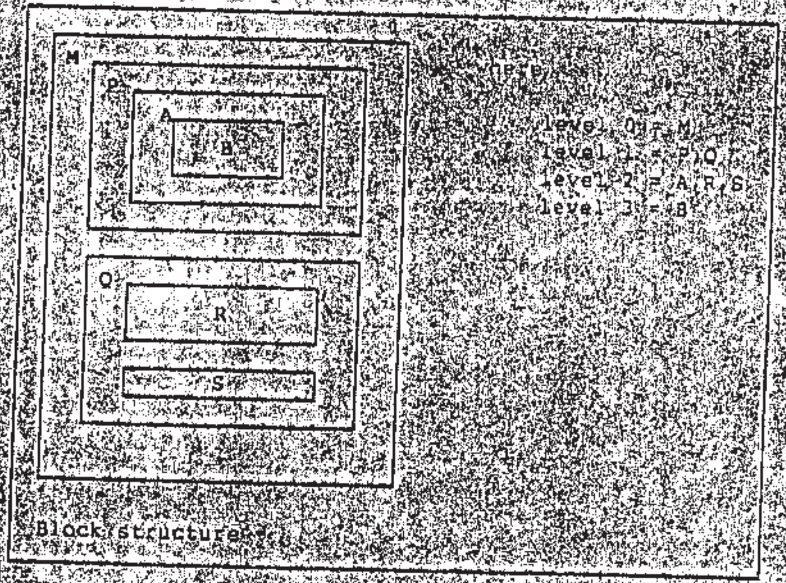
Block structure

QL Pascal 68000 source code is a collection of identically structured units known as blocks. Each block has the form

```
<block heading>
    [<declaration>]
    [<definition>]
BEGIN
    [<statement>]
END<[illegible]>
```

Each block has a required heading, an optional declarations and definitions section and, surrounded by the reserved words BEGIN and END, zero or more statements. Statements are language specific syntactic constructs used to control and perform action within a program. Blocks are discrete program chunks that cannot overlap with other blocks, but can fully reside within other blocks which, in turn, can reside in other blocks and so on. This is known as block nesting. In nested blocks the outermost block is the main block and block nesting can be used to create many block levels (see Fig 1).

Figure 1



Thus in a program the outermost block level is that of the main control or program block.

A PASCAL Program

A complete program contains one or more blocks. It must contain at least the program block to which control is first passed at run-time. The program block has the following form:

```
PROGRAM <program name> {<program-parameters>}";"

{<declaration>}

{<definition>}

BEGIN

{<statement>}

END"."
```

A program starts with the reserved word PROGRAM followed by its identifying program name, which has no further use within the program, and any optional program parameters (these are described in Sections 7.5 and 7.6). The program block definition ends with a full stop. A program with zero number of statements will, of course, do nothing.

Other blocks are defined as **procedures** and **functions**, to which program control can be passed. Procedures and functions can contain nested procedures and functions. Procedures and functions can be thought of as subprograms which are associated with identifiers declared in the declarations and definitions section of a block. A function is distinct from a procedure in that the identifier used to declare the function is also associated with the result of the function subprogram.

Apart from reserved words and certain predefined identifiers each identifier, which can relate to a procedure, a function, a label or an item of data must be declared before it is referenced from within a program. The initial definition or declaration of an identifier constitutes its **defining point**. An identifier may only be referenced from within what is known as the **scope** of the identifier.

Scope

The scope of an identifier is the set of all blocks where a valid reference to an identifier can be made. The normal scope or **region** of an identifier is anywhere inside its defining block starting from its defining point. If an identifier name is also used to define an identifier within a nested procedure then the outer block identifier becomes inaccessible from the inner block. Thus an identifier's scope can be smaller, but never larger, than its region. An identifier defined in the program block is said to be global, as its region is the entire program. An identifier defined in a nested block is said to be local to its defining block.

Declarations and Definitions

In every block, the declarations and definitions section consists of:

```
{<label declaration>}

{<constant definition>}

{<type definition>}

{<variable declaration>}

{<procedure or function declaration>}
```

These subsections must be specified in the above relative order.

Label declarations

QL Pascal 68000 statements can be labelled as target destinations for program control transfer by GOTO statements. Labels are predeclared syntactically:

```
LABEL n1{,n2....,nn}";"
```

where n1...nn are distinct integers in the range 0 to 9999. Labels may only prefix a single statement in the block that immediately contains its declaration and not in any block within that block. (see GOTO statement in Section 5.4.3).

Constant definitions

If the value of data associated with an identifier is to remain fixed for the duration of program execution, the identifier can be defined as a constant as opposed to being declared as a variable. Constants are defined syntactically:

```
CONST <constant definition> ";"
                    {<constant definition> ";"}
```

Constants are further discussed in the Section 3.2.

Type definitions

The form of data used in QL Pascal 68000 programs can be specified as type definitions:

```
TYPE <type definition> ";" {<type definition> ";"}
```

Through type definitions, QL Pascal 68000 provides easy manipulation of complex and flexible data structures. Type definition is discussed in Section 4.

Variable declarations

The allocation of data for use in QL Pascal 68000 programs is specified as variable declarations:

```
VAR <variable declaration> ";"
                    {<variable declaration> ";"}
```

Variable declaration is discussed with type definition in Section 4.

Procedure and Function declarations

Procedures and functions are declared syntactically:

```
   <procedure or function heading>
   [<declaration>]
   {<definition>}
   BEGIN
      {<statement>}
   END ";"
```

Procedures and functions are fully described in Section 6.

Labels are not treated as identifiers and are not fully subject to normal scope rules. Also, attention is drawn to the FOR statement (see Section 5.3.1) regarding referencing identifiers under normal scope rules.

Statements

The desired action on the declared data is effected by the correct syntactic and logical use of statements, which describe how the related data is to be manipulated. Statements are described in Section 5.

The following is a simple program example designed to encapsulate this introduction to QL Pascal 68000:

```
PROGRAM Introduction(Input, Output);

CONST Pi = 3.1416;

TYPE Length = REAL;

VAR Radius, Diameter : Length;

   FUNCTION AreaOfCircle : REAL;

      BEGIN

         AreaOfCircle := Pi * Radius * Radius

      END;

BEGIN

  WRITELN('Enter circle diameter ');

  READLN(Diameter);

  RADIUS := Diameter / 2.0;

  WRITELN('The area of your circle is ', AreaOfCircle)

END.
```

3.2 Language vocabulary and data

The basic QL Pascal 68000 vocabulary consists of the special symbols and reserved words itemised in Section 3.1 and certain predefined or standard identifiers. The vocabulary is extended by programmer defined identifiers. Reserved words and standard identifier names cannot be used to define identifiers. All these tokens are separated from each other by using any combination of the following separators:

i) any number of blank characters
ii) any number of 'end of line' characters
iii) any number of comments

Comments

Comments can be inserted into QL Pascal 68000 programs by enclosing any desired sequence of symbols, excluding the symbol "}", by a pair of curly brackets.

```
"{"<any symbol sequence not containing "}">"}"
```

If necessary, "(*" can be used in place of "{" and "*)" can be used in place of "}". Comments are generally applied to clarify the intended action of a program.

```
e.g.

    {this is a comment}

    (*this is also a comment*)
```

Standard identifiers

There are a number of standard identifiers which are predefined for immediate use in QL Pascal 68000 programs at all block levels. They are described at relevant points throughout this manual but the following is a list of their names:

```
ABS      ARCTAN    BOOLEAN   CHAR      CHR

COS      DISPOSE   EOF       EOLN      EXP

FALSE    GET       INPUT     INTEGER   [illegible]

MAXINT   NEW       ODD       ORD       OUTPUT

PACK     PAGE      PRED      PUT       READ

READLN   REAL      RESET     REWRITE   ROUND

SIN      SQR       SQRT      SUCC      TEXT

TRUE     TRUNC     UNPACK    WRITE     WRITELN
```

Data

Data is the name given to all that is operated on by a computer. Ultimately all data are represented in the machine as sequences of binary digits.

For example, QL Pascal 68000 source code is a collection of data for input to the QL Pascal 68000 compiler.

This section continues with a description of fundamental data literals used and understood by QL Pascal 68000 programs.

Numbers

Decimal notation is used for numbers. A number can be positive or negative and cannot contain embedded blanks or commas. Numbers can be specified as either **integers** or **real numbers** which are each processed differently at run-time.

Integers

Whole numbers in the 32 bit range

```
-2147483648 to +2147483647
```

can be treated as integers. Formally integers are represented by:

```
<signed-integer> = ["+"|"-"]<unsigned-integer>

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

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

The sign can be omitted for positive integers.

When performing computations in a program using integers, unpredictable results can occur if possible intermediate values are not in the range specified for integers. The following are all examples of valid integers:

```
0     -1     +5     -007     6789300
```

Real Numbers

These take the form:

```
["+"|"-"]<unsigned-real>
```

where

```
[illegible] <unsigned-integer> "."
                    <fractional-part>
                    [E <scale-factor>]
[illegible]

<unsigned-real> ::= <unsigned-integer> E <scale-factor>

<fractional-part> ::= <digit-sequence>

<scale-factor> ::= <signed-integer>
```

The sign may be omitted for positive real numbers. The construct "E" <scale-factor> is used to represent the preceding number times ten to the power of <scale-factor>, <scale-factor> being an optionally signed whole number of one or two digits. When specifying a real number, this construct must be included if the number is specified without the use of a decimal point. It is optional when using a decimal point. A decimal point must have at least one digit either side of itself.

The following are all examples of valid real numbers:

```
1E5   [illegible]26E-1   [illegible]3.0   12.34E+20
```

whereas the following are all examples of invalid numbers:

```
3.[illegible]56[illegible]   [illegible]8[illegible]E9[illegible]   [illegible]E2[illegible]   1.0E100
```

This numeric representation allows numbers to be represented in many ways. Thus for example,

the integer 123456 can be specified as any of the following real numbers:

```
123456.0   123456e0   [illegible]   1.23456E5
```

Strings

Sequences of characters enclosed by single quote marks are referred to as strings. To include a quote mark, two quote marks are specified in the string. The following are all examples of literal strings:

```
'a'   '1'   ''''   'begin'   'can''t'   'string'
```

Constant definition

Named constant values and literals are defined as constant definitions:

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

<constant> can be any of the forms just described. In a constant definition part of a block, the identifiers must be distinct, if necessary, be used in place of <constant>. For example,

```
CONST

     topnum = 50;

     lownum = -topnum;
```

The following is an example of a valid constant definition part of an QL Pascal 68000 program which also incorporates the use of comments.

```
CONST

    message = 'welcome';       {this is a string}

    DIMENSION = 100;           {this is an integer}

    Factor = 5.4E-2;           {this is a real number}

    BlankString = '';          {this is a string}
```

Defined constants are identifiers that conform to normal scope rules. They can also be referenced in type definitions for the purposes of specifying subranges or array bounds (see Sections 7.1 and 7.2).

MAXINT

QL Pascal 68000 provides one predefined constant as the identifier MAXINT. It represents the largest positive integer of the 32-bit range (see integers).

```
MAXINT   -   represents the positive integer 2147483647

-MAXINT  -   represents the negative integer -2147483647
```

Chapter 4.

Type definitions and variable declarations

All static data used in a program are specified as variable declarations. The form and range of data are described as type definitions.

All data must be declared before use by program statements in the variable declaration parts of program blocks. A data type definition can reside alongside the data declaration or reside in a type definition part before data declaration. Using type definitions is to be recommended, for other than predefined data types, in order to minimize problems that may be encountered due to data type mismatches. It also aids in the production of a more understandable source code program.

The **Type definition** part of a block is:

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

and the **Variable declaration** part of a block is:

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

where <type> in both parts must be equal to either an QL Pascal 68000 provided type or <identifier> of a previous type definition whose scope contains the type definition or variable declaration. The block actually containing the type definition or variable declaration and all blocks nested within, constitute the region of <identifier>.

All variables whose identifiers are declared in the Variable Declaration part of a block, except those listed as program parameters, shall be totally undefined when execution of the statement part of their block commences. See appendix C.

Data types fall into two categories : Simple types and Structured [illegible]

4.1 Simple types

Simple types are collections of elementary and [illegible] data items. [illegible] four simple types provided by QL Pascal 68000 for immediate [illegible] variable declarations:

- Boolean
- integer
- char
- real

These data types can be separated into two categories: ordinal and real.

Ordinal Types

An ordinal type is characterized by being enumerable; ordinal type values can be numbered, and compared for equality and relative position. Thus a subrange of a full ordinal type range can be defined; this definition is known as a subrange type. Char and Boolean are ordinal types and type integer is of course an ordinal type in the purest sense. Type Boolean belongs to the final class of ordinal type known as an enumerated type. An enumerated type is a collection of programmer specified identifiers; for type Boolean the identifiers are TRUE and FALSE.

Type real

Type real refers to real numbers as discussed in section 3.2. The range of real numbers is bound by the implementation and their machine representation is specifically designed for storage efficiency by virtue of treating real numbers as having two distinct parts - the digits of the number itself and the exponent (the part beginning with "E").

The provided simple types can be renamed in type definitions if required.

4.1.1 Type BOOLEAN

```
<type> = BOOLEAN
```

A Boolean type has two values denoted by the predefined identifiers FALSE and TRUE. Comparison between Boolean identifiers accords with the ordinal values of FALSE and TRUE which are 0 and 1 respectively.

Logical operators

The following logical or Boolean operators can only be applied to Boolean operands and yield Boolean values:

```
AND  -  logical conjunction

OR   -  logical disjunction

NOT  -  logical negation
```

Relational operators

Each of the relational operators when applied to the various permitted operand types (see sub-section on expressions in section 5) yields a Boolean value:

```
=    -    equality

<>   -    inequality

<    -    less than

>    -    greater than

<=   -    less than or equal

>=   -    greater than or equal
```

IN contained in

Predefined logical functions

The following predefined functions yield Boolean values:

ODD(x) TRUE if integer x is odd, otherwise FALSE

EOLN(f)
EOF(f) These functions are concerned with file handling and are dealt with in sections 7.5 and 7.6

4.1.2 Type INTEGER

`<type> = INTEGER`

The following arithmetic operators yield an integer value when applied between integer operands:

	multiply
DIV	divide and truncate (the value is not rounded) division by zero constitutes an error
MOD	the result of i MOD j is i-(n*j) for integer n such that 0 <= (i MOD j) < j

A zero or negative right hand operand constitutes an error

	add
	subtract

Add and subtract can also be applied to single operands.

The relational operators = <> < > <= >= yield a Boolean value when applied between integer operands.

The following predefined functions yield integer values:

ABS(x)	gives the absolute value of integer x
SQR(x)	gives the squared value of integer x
TRUNC(x)	gives the integer value to the left of the decimal point for real value x
ROUND(x)	gives the rounded integer value of real value x thus if x >= 0 the result is TRUNC(x + 0.5) if x < 0 the result is TRUNC(x - 0.5)

and the ordinal functions, when applied to an integer:

SUCC(x)	yields the next integer (x+1)
PRED(x)	yields the preceding integer (x-1)
ORD(x)	yields the ordinal integer associated with the value of x, where x is an ordinal type variable.

(the integer range or subrange must be borne in mind when applying SUCC, PRED, SQR, TRUNC and ROUND to avoid run-time errors)

The description of the ordinal function CHR is included in the sub-section on Type CHAR (see section 4.1.4).

It follows that for a value c of type char and a value i of type integer:

```
CHR(ORD(c)) = c and
ORD(CHR(i)) = i
```

Relational comparisons between values of type char correspond to the relationships between the ordinal numbers of the values. Thus

If ORD(c1) < ORD(c2) where c1 and c2 are values of type char then c1 < c2.

This applies to all of the relational operators.

The functions PRED and SUCC can be applied to values of type char, yielding results in accordance with the ASCII character set collating sequence.

```
PRED(c) = CHR(ORD(c) - 1)

SUCC(c) = CHR(ORD(c) + 1)
```

for a value c of type char.

NOTE: The small range of the character set must be borne in mind when applying these functions to values of type char.

The following is a program example to illustrate full type definitions and variable declarations.

```
PROGRAM TypeAndVar(output);

 TYPE Degrees = REAL;

      NumberofPeopleinAttendanceattheMeeting = INTEGER;

      IsItMorning = BOOLEAN;

      ALetteroftheAlphabet = CHAR;
```

```
 VAR  Temperature,Hotness,Coolness : Degrees;

      IsItAfternoon : BOOLEAN;

      AM : IsItMorning;

      C : ALetteroftheAlphabet;

      Letters : CHAR;

      WholeNumbers : INTEGER;
      SundayGathering :
            NumberofPeopleinAttendanceattheMeeting;

      Profit,Loss,Costs,Margin : REAL;

BEGIN



END.
```

Structured types

[illegible] data types can be defined in QL Pascal 68000 but must [illegible] collection of simple types. The permitted [illegible] types are as follows:

- sets described in section 7.1
- arrays described in section 7.2
- records described in section 7.3
- pointers described in section 7 [illegible]
- files described in section 10

The [illegible] type can be regarded as a non-ordinal simple type.

Chapter 5: Statements

5.1 Control and action in PASCAL programs

Statements provide control and action within a program. Statements fall into two categories:

i) **simple and compound statements**

ii) **structured statements**

The empty statement, procedure invocations and assignment statements (which encompass function invocations) constitute simple statements. Compound statements are essentially sequences of simple and structured statements. Structured statements relate more to program control consisting of conditional and repetitive statements.

Compound statement

Syntactically this is represented by:

```
BEGIN

    {<statement>}

END[";"|"."]
```

A compound statement comprises a collection of component statements (simple or structured) surrounded by the reserved words BEGIN and END. It is executed as a sequence of executions determined by the nature of the component statements as they are written. The statement body of a program block has the form of a compound statement which ends with "." for the main block or ends with ";" for a subprogram block.

e.g.

```
PROGRAM CompoundStatement(output);
VAR result : INTEGER;
BEGIN
  result := 9;
  WRITELN(result);
  result := SQR(result);
  WRITELN(result)
END.
```

produces output

```
[illegible]
```

Empty statement

The previous example also illustrates that a semicolon separator is optional before the final END of a compound statement. If a semicolon is used then an empty statement is said to exist between the semicolon and the END. The empty statement is a simple statement which is harmless and does nothing. It is possible to insert semicolons mistakenly which may result in compilation or execution errors that can be difficult to locate.

Structured statements

Sequence control within a program can be effected in two different ways:

i) by using statements that control repetition of some specified action

or

ii) by using branching statements that can select control or transfer control

6.2 ASSIGNMENT statement

This is the most fundamental action statement in QL Pascal 6800

Its form is

```
<variable> := <expression>
```

:= is known as the **assignment operator**, distinct from the relation operator "=". Assignment specifies that a newly computed value i assigned to <variable>, which is an identifier (or a function designat – see section 6) declared in a variable declaration part such that t assignment statement is within the scope of the identifier.

e.g.

```
x := 7; {an example of the simplest form of assignmen

Result := 9.6 * 10.8 * 6.1 - Offset;
```

The new value is obtained by evaluating <expression>.

Expressions

An expression is a rule for calculating a value and is made up fr identifier or literal operands, operators and identifiers to inve functions (function designators – see section 6). It is an error for undefined variable-access to appear in an expression. See appendix The evaluation is subject to operator precedence rules but beyond th proceeds from left to right. (However, see appendix C.)

5.2.1 Operators

[illegible] precedence

[illegible] four levels of precedence.

[illegible] precedence is given to the logical operator NOT. Precedence is [illegible] the logical operator AND and the multiplying operators [illegible] precedence is given to the logical operator [illegible] adding operators [illegible] lowest precedence is given to the [illegible] operators.

[illegible] enclosed within parentheses are evaluated regardless of [illegible] succeeding expressions.

[illegible]

[illegible] rules

Operations between variables can only take place if the variables [illegible] compatible types.

[illegible] rules

For an operation between a variable of type T1 and a variable of type [illegible] compatibility is summarised as follows:

T1 and T2 are the same type.

Ordinal type T1 is a subrange of T2 (or *vice versa*) or both are subranges of the same host ordinal type.

Set types T1 and T2 are compatible if the ordinal base types are compatible and if both or neither are packed (see 6.1).

T1 and T2 are string types with the same number of components (see 6.2).

Assignment rules

Assignment is possible to variables of any type, with the exception of type file (see section 7.5). Assignment is only possible if the variable type and the expression yielded value are assignment compatible.

Assignment compatibility rules

For an expression value of type T1 and variable of type T2 assignment compatibility is summarised as follows:

T1 and T2 are the same type.

T2 is of type REAL and T1 is of type INTEGER or subrange of INTEGER but not *vice versa*.

T1 and T2 are compatible ordinal or enumerated types and the expression value is valid for type T2.

T1 and T2 are compatible SET types and every set member given by the expression is contained by the base type of T2 (see section 7.1).

T1 and T2 are compatible string types (see array types in section 7.2).

Examples of valid assignment statements

```
count := count + 1;

area := radius * radius * PI;

perimeter := 2 * (length + width);

RSquared := SQR(r);

Z1 := SIN(x1) + COS(y1);

Margin := SellingPrice - Costs;

Correct := Answer = RightAnswer;
```

Operator summary

Table 1 : Monadic Arithmetic Operators

operator	*operation*	*type of operand*	*type of result*
+	identity	*integer* *real*	*integer* *real*
-	sign inversion	*integer* *real*	*integer* *real*

Table 2 : Dyadic Arithmetic Operators

operator	*operation*	*type of operands*	*type of result*
+	addition	*integer or real*	*integer if both operands are integer otherwise real*
-	subtraction	*integer or real*	
*	multiplication	*integer or real*	
/	division	*integer or real*	*real*
div	truncated division	*integer*	*integer*
mod	modulo	*integer*	*integer*

Table 5.1: Boolean Operators

operator	type of operation	type of operands	type of result
not	negation	*boolean*	*boolean*
or	disjunction	*boolean*	*boolean*
and	conjunction	*boolean*	*boolean*

Table 5.[illegible]: Set Operators

operator	type of operation	type of operand	type of result
+	union	any canonical set-of-T type	same as the operands
-	set difference		
*	set intersection		

Table 5.[illegible]: Relational Operators

operator	type of operand	type of result
= <>	any simple, pointer, or string type, or canonical set-of-T type	*boolean*
< >	any simple or string type	*boolean*
<= >=	any simple or string type, or a canonical set-of-T type	*boolean*
in	left operand: any ordinal type T right operand: a canonical set-of-T type	*boolean*

5.3 Repetition

There are three forms of repetition statements:

1. FOR statement
2. WHILE statement
3. REPEAT statement

5.3.1 FOR statement

Syntactically this is:

```
FOR <control-variable> :=
    <expression1> TO | DOWNTO <expression2> DO
    <statement>
```

Upon execution of the FOR statement the values of <expression1> and <expression2> are evaluated (<expression1> being evaluated first) and stored to determine the number of repetitions of <statement>. <statement> can be any simple, compound or structured statement and is executed for each increment of <control-variable> when using the TO option and for each decrement when using the DOWNTO option. The value of <control-variable> is left undefined at termination of the FOR statement even if <statement> is not executed. The first part of a FOR statement can be thought of as being composed of two assignment statements and as such, <control-variable>, <expression1> and <expression2> are subject to assignment compatibility rules as well as the following:

(i) <control-variable> must be declared as an identifier of any ordinal type. Therefore it cannot be declared using type real and it cannot be a component of a structured variable or a variable accessed through a pointer (see section 7.4).

(ii) <control-variable> must be declared as an identifier in the block that immediately contains the FOR statement and not in any outer blocks.

(iii) <control-variable> must not be 'threatened' within the FOR statement action or in any blocks local to the block immediately containing the FOR statement. 'Threatened' action constitutes any of the following:

(a) An ordinary assignment to <control-variable>.

(b) Passing <control-variable> as a variable-parameter to a procedure or function (see section 6).

(c) READ or READLN (see sections 7.5 and 7.6) calls with <control-variable> as a parameter.

(d) <control-variable> acting as a <control-variable> for another FOR statement.

(iv) If <expression1> is greater than <expression2> when using the TO option, or <expression1> is less than <expression2> when using the DOWNTO option, <statement> will not be executed. If <expression1> is equal to <expression2>, for either the TO or DOWNTO options, <statement> will be executed once.

e.g.

```
FOR i := 11 TO 10 DO <statement>

FOR i := 10 DOWNTO 11 DO <statement>
```

<statement> in the above examples will not be executed.

```
FOR i := 10 TO 10 DO <statement>

FOR i := 10 DOWNTO 10 DO <statement>
```

here <statement> will be executed exactly once in each case.

e.g.

```
PROGRAM forloop(output);
VAR
  i, j:INTEGER;
BEGIN
  j:=4;
  FOR i:=1 to j+2 DO
    BEGIN
      WRITE(i);
      WRITE(' ')
    END;
  WRITELN
END.
```

5.3.2 WHILE statement

Syntactically this is:

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

<expression> must yield a value of type Boolean and is evaluated before each possible execution of <statement>. <statement> can be any simple, compound or structured statement and is executed each time <expression> yields Boolean value TRUE. Execution of the WHILE statement terminates upon <expression> yielding Boolean value FALSE. <statement> can, therefore, be executed zero or more times. The following are examples of valid WHILE statements:

```
WHILE BankBalance > 50 do
  BEGIN
   SupplierAccount:=SupplierAccount + 50;
   BankBalance:=BankBalance - 50
  END;

WHILE positive > 0 DO
  positive:=positive-1;
```

5.3.3 REPEAT statement

Syntactically this is:

```
REPEAT
   <statement>
UNTIL <expression>;
```

<expression> must yield a value of type Boolean and is evaluated after each execution of the statement body. The statement body can consist of any number of any simple or structured statements. It can be a compound statement delimited by the reserved words REPEAT and UNTIL although BEGIN and END may also be included as delimiters if preferred. The statement body is executed at least once, and repeatedly executed each time <expression> yields Boolean value FALSE. Overall REPEAT statement execution is terminated when <expression> yields Boolean value TRUE.

```
REPEAT
   PurchaseAccount:=PurchaseAccount + 50;
   BankBalance:=BankBalance - 50
UNTIL BankBalance < 50;
```

is an example of a valid REPEAT statement (which may give rise to an overdraft!).

5.4 Branching statements

Control selection and transfer in a program is effected using IF and CASE statements. Control transfer alone, which can be brought about by procedure and function invocations in statements can also be effected using the GOTO statement.

5.4.1 IF statement

An IF statement can take one of two syntactic forms:

```
i)  IF <expression> THEN <statement>

ii) IF <expression> THEN <statement>
                    ELSE <statement>
```

<expression> must return a value of type Boolean. In form i) <statement> is executed when <expression> yields a value of TRUE, for <expression> value FALSE, execution continues at the point immediately follwing the IF statement. In form ii) the first <statement> is executed for <expression> value TRUE and the second <statement> is executed for <expression> value FALSE. <statement> is any simple, structured or compound statement. Form i) is really an abbreviation of form ii) with the second <statement> being the empty statement.

In the following complex expression:

```
IF (<expression1> (AND|OR) <expression2>)
      THEN <statement1>
      ELSE <statement2>
```

<expression1> will be evaluated first followed by the evaluation of <expression2> followed by the application of the logical operator before the actioning of the IF statement. Expression evaluation complies with the precedence rules described earlier in this section. For complex expressions, it may then be more practical to build a nested IF statement construct.

```
IF <expression1> THEN
```

```
IF <expression2> THEN <statement>
```

This may have the advantage of streamlining the IF statement [illegible] <expression2> will not be evaluated if <expression1> [illegible] value FALSE. Nesting within IF statements can exist to a considerable degree. In such cases each ELSE is paired with the nearest unpaired THEN. If necessary BEGIN / END pairs can be used to ensure the intended IF statement nested construct action.

[illegible] construct [illegible]

```
IF <expression1> THEN
   IF <expression2> THEN
      <statement1> ELSE
      <statement2>
```

[illegible]

Without the use of a BEGIN / END pair the action construct taken would be that of form i).

NOTE. A semi-colon must not be inserted before the reserved word ELSE.

USEFUL HINT Conditional assignment of Boolean variables to Boolean values as in

```
IF x = y THEN Thesame := TRUE
         ELSE Thesame := FALSE;
```

can be actioned using simpler and more efficient assignment statements of the form

```
<Boolean identifier> := <expression>
```

thus:

```
Thesame := x = y;
```

has the same effect as the preceding IF statement.

The following is an example of a nested IF statement construct to test for several positive values of the variable 'number'

```
IF number = 10 OR number = 20 THEN
<action1>
ELSE
 IF number = 30 THEN
  <action2>
  ELSE
   IF number = 40 THEN
    <action3>
```

Testing for several possible values of a variable as in the above example, can be accomplished more concisely by use of the CASE statement.

5.4.2 CASE statement

Syntactically this is

```
CASE <expression> OF
<case-label-list> ":" <statement>
{";" <case-label-list> ":" <statement>}
END
```

where

```
<case-label-list> ::=
     <label-constant> {"," <label-constant>}
```

<expression> is evaluated and then acts as the selector for comparison with the <label-constant> in <case-label-list>. <expression> value and <label-constant> are of any ordinal type. Upon a precise match of <expression> value and <label-constant>, the <statement> corresponding to the <case-label list>, of which the matching <label-constant> is part, is executed. All occurrences of <label-constant> in any <case-label list> must be distinct and unique. It is an error if there is no match of <expression> value and <label-constant>. Upon completion of execution of a selected <statement>, program execution continues at the point immediately following the CASE statement (unless <statement> incorporates a GOTO statement). <statement> can be any simple, structured or compound statement. <label-constant> cannot be an identifier.

```
        CASE number OF
          10,20:<action1>;
             30:<action2>;
             40:<action3>
        END
```

This example achieves the same results as the example at the end of the discussion about the IF statement.

NOTES

i) Case label constants are not labels as declared in a label declaration part of a block; they cannot be used as target destinations for GOTO statements.

ii) Although the <case-label-list> may contain a <label constant> to which the <expression> may never be evaluated, its inclusion in the <case-label-list> is to be discouraged as it serves no useful purpose.

5.4.3 GOTO statements

Syntactically this is

```
GOTO <label>[";"].
```

This states that program control is to be unconditionally transferred to the simple or structured statement prefixed by <label>. <label> is any whole number in the range 0 to 9999. Target destination syntax:

```
<label> ":" <statement>
```

Each label must be predeclared in the label declaration part of a block (see section 3.1) and can prefix a single statement in only that block and not in any blocks local to that block.

A GOTO statement can only cause a branch to certain statements and the placement of labels must accord with the rules governing the target destination of a GOTO statement which states that the target destination can be any of the following:

i) the statement that contains the GOTO

iii) another statement in the [illegible] the GOTO is part of or a statement in a [illegible] that contains the GOTO's [illegible]

iii) another statement in any block that contains the GOTO, as long as that statement is not part of the action of a structured statement (aside from the compound statement that forms a block's statement part), i.e. the target label must be at the outermost level of a structured statement.

If, as in Rule (iii), a GOTO statement is used to jump to a statement in a containing block, then the block containing the GOTO statement and all other activated nested blocks contained by the target block becomes deactivated. (Block activation is described in section 6).

[illegible]

```
PROGRAM [illegible]Gotos(input);
LABEL 2,9999;
VAR [illegible] : CHAR;
PROCEDURE [illegible];
BEGIN
[illegible]
GOTO 9999
[illegible]
```

[illegible]

Chapter 6: Subprograms

6.1 Procedures

Procedures and functions are subprogram blocks that reside within the main program block to which program control can be passed. Program development can start with the main program block and gradually progress with the introduction of procedures and functions when required. Repeated code can be defined as a subprogram block. A subprogram can be used to isolate source code that is very complex or is likely to require periodic amendment. A function differs from a procedure in that it returns a result that is associated with the identifier that is used to define the function. Procedures and functions must be declared at the end of the definitions and declarations part of all blocks - the program block or a procedure or function block.

A Procedure declaration

A procedure declaration has the form:

```
<procedure-declaration> ::=
PROCEDURE <procedure-identifier>
        [<formal-parameter-list>] ";" [<directive> ";"]

[<definitions>]

[<declarations>]

BEGIN

[<statement>]

END
```

The procedure definitions and declarations are local to the procedure. Global definitions and declarations are available for reference and alteration by the procedure (except as control variables in FOR statements, see section 5) unless excluded by the use of global identifiers as identifiers for local definitions and declarations. The statement can be any simple or compound or structured statement. Local variables are active only for the period of activation of the procedure.

Activation

A subprogram becomes activated when it is invoked and becomes deactivated upon return to its calling point. Thus if an invoked subprogram invokes a nested subprogram a chain of active subprograms exists.

A Procedure call

A procedure is invoked by call in a procedure statement.

```
<procedure-statement> = <procedure-identifier>
                 [<actual-parameter-list>][";"]
```

The procedure identifier is specified, followed by any actual parameters required as specified in the formal parameter list of the procedure declaration. For order of evaluation of the parameters see appendix C.

Recursion

A procedure (and a function) can call itself from its own statement body, and in so doing becomes a recursive procedure. This is best explained using an example:

```
PROGRAM invert (input,output);

 PROCEDURE stack; {example of recursion}

   VAR letter:CHAR;
```

```
   BEGIN
    READ(letter);
    IF NOT EOLN THEN stack;
                 {check no more letters in input line}
    WRITE(letter);
   END;

BEGIN
 stack;
END.
```

produces for an input line of 'to illustrate recursion'

```
noisrucer etartsulli ot
```

(i.e a reversal of the input line)

6.2 Functions

A function declaration has the form:

```
<function-declaration> =
FUNCTION <function-identifier>
     [<formal-parameter-list>]":"<result-type>";"
                                   [<directive>";"]


  [<definitions>]

  [<declarations>]

  BEGIN

  [<statement>]

  END
```

The function definitions and declarations are local to the function. Glob definitions and declarations are available for reference by the functio (except as control variables in FOR statements (see section 5) unles excluded by the use of global identifiers as identifiers for local definitio and declarations. The statement can be any simple, compound [illegible] statement. Local variables are active only for the period of [illegible] of the function.

A function declaration is like that of a procedure declaration with the exception of a result type associated with the function identifier. The result type is the identifier of an already defined type; a new type cannot be defined in a function declaration. The result type can be any simple or pointer type. The intention of a function is to return a single result type, although like a procedure, results can be returned through the formal parameter list using VAR parameters.

A function call

A function is invoked by the appearance of a function designator in [illegible]

[illegible]

6.3 Formal parameter list

There are four kinds of formal parameters that can be specified in a procedure or function declaration.

```
<formal-parameter-list> =
              "(" <formal-parameter-section>
                {";" <formal-parameter-section>} ")"

<formal-parameter-section> =
              <value-parameter-section> |
              <variable-parameter-section> |
              <procedural-parameter-section> |
              <functional-parameter-section>
```

(i) value parameters are similar to local variable declarations in subprogram blocks and are initialized when the subprogram is invoked. The action of the subprogram does not affect the actual parameter expressions that provide the value parameters at subprogram call time.

(ii) variable parameters are again similar to local variable declarations in subprogram blocks but an assignment, within the subprogram, to a variable parameter is equivalent to an assignment to the parallel actual parameter specified in the subprogram call.

(iii) procedural parameters are similar to local procedure declarations in subprogram blocks, with the actual procedures declared elsewhere.

(iv) functional parameters are similar to local function declarations in subprogram blocks, with the actual functions declared elsewhere.

The number and type of the actual parameters specified in the subprogram call must be the same as, and must be specified in the same order as, the number and type of the parameters specified in the subprogram itself, this applies to all possible combinations of subprogram parameters. Value and variable parameters must be specified using already existing type definitions. Value and variable parameter identifiers cannot be used as identifiers for definitions and declarations within the subprogram block. For order of evaluation of the parameters see appendix C.

Value parameters

```
<value-parameter-specification> =
                   <identifier> {"," <identifier>} ":"
                                 <type-identifier>
```

The initial value of a value parameter is supplied by an actual parameter. The actual parameter that corresponds to a value parameter can be any expression that is assignment compatible with the value parameter. An assignment to a value parameter does not alter the value of the actual parameter of the subprogram call. File type variables (or structured variables with a file-type component) cannot be passed as value parameters.

e.g.

```
PROGRAM valpars(output);
  VAR a, b: integer;

PROCEDURE nochange(a, b: INTEGER);
  {globals a and b excluded from nochange}

BEGIN
  [illegible]
  WRITELN(a, [illegible]
  {a and b effectively local variables}
END;

BEGIN
  a := [illegible]
  [illegible]
  WRITELN(a, b);
  nochange(a, b);
  {globals a and b passed as actual parameters}
  WRITELN(a, b)
  {upon return globals a and b remain unchanged}
END.
```

produces output

```
1 5
9 [illegible]
1 5
```

Variable parameters

```
<variable-parameter-specification> =
                   VAR <identifier> {"," <identifier>} ":"
                                     <type-identifier>
```

The reserved word VAR must be repeated with each additional type of variable parameter. The actual parameter that corresponds to a variable parameter must be a variable access and not a value, such as a constant or function call; thus variable parameters act as synonyms, local to subprograms, for accessing variables declared elsewhere, and changes to variable parameters amount to changes to the corresponding actual parameters. The following four restrictions apply to variables passed to variable parameters:

- The actual parameter must possess the same type as its corresponding variable parameter.

- The actual parameter may not denote a component of a packed variable (although a packed variable may be passed as a parameter).

- The actual parameter may not denote a field that is the selector of a record's variant part.

- If a file buffer variable f↑ is passed as the argument of a variable parameter, it is an error to modify the value of the file f.

e.g.

```
PROGRAM varpars(output);

VAR radius:INTEGER;

  PROCEDURE cube(VAR r:INTEGER);

   BEGIN
    r:=r*r*r
   END;

  BEGIN
   radius:=5;
   WRITELN(radius);
   cube(radius);
   WRITELN(radius)
  END.
```

produces

[illegible]

Actual parameters are passed to variable parameters when they require modification by the called subprogram. Although passing actual parameters to value parameters is more secure, it is more demanding on [illegible] copy is made of the actual parameter to act as the value parameter so, for example, passing an array as a value parameter will require an extra amount of storage equal to the size of the array.

Procedural and functional parameters

A procedural parameter is a synonym, local to the called subprogram, for a procedure declared elsewhere.

```
<procedural-parameter-specification> =
                          <procedure-heading>

<procedure-heading> = PROCEDURE
            <identifier>[<formal-parameter-list>]
```

Likewise a functional parameter is a synonym, local to the called subprogram, for a function declared elsewhere.

```
<functional-parameter-specification> =
                          <function-heading>

<function-heading> = FUNCTION
            <identifier>[<formal-parameter-list>]
                                    ":"<result-type>
```

The identifiers in the formal parameter list have no meaning or application within the called subprogram. Procedural and functional parameters are not accompanied by their own actual parameters - the actual parameter identifiers are dummy identifiers. However calls to the subprograms denoted by the procedural and functional parameters, from within the called subprogram, must be accompanied by the actual parameters specified by the formal parameter lists of the procedural and functional parameters in accordance with the rules previously described.

e.g.

```
PROGRAM procfuncpars(output);

VAR i:INTEGER;

 PROCEDURE square(VAR j:INTEGER);
  BEGIN
   j:=j*j
  END;

 PROCEDURE cube(VAR k:INTEGER);
  BEGIN
   k:=k*k*k
  END;

 PROCEDURE apowerof(var l:INTEGER;
                        PROCEDURE power(var m:INTEGER));
  BEGIN
   power(l)
  END;
```

```
BEGIN
  i:=5;
  WRITELN(i);
  Apowerof(i,square);
    {only procedural parameter identifier specified}
  WRITELN(i);
  i:=3;
  WRITELN(i);
  Apowerof(i,cube);
    {only procedural parameter identifier specified}
  WRITELN(i)
END.
```

produces

```
  5
 25
  3
 27
```

6.4 The FORWARD directive

In the declaration of a procedure or function, the forward directive can be specified. It allows a forward reference whenever a subprogram identifier must appear in advance of its declaration. This directive has been provided to cater for mutually recursive subprograms. The subprogram identifier and its formal parameter lists (and result type if it is a function) are specified followed by the reserved word FORWARD; the subprogram block can then be declared anywhere beyond this point provided the declaration is nested in the same region and nested at the same level as the FORWARD specification. The block declaration is headed by the relevant subprogram reserved word followed by just the subprogram identifier.

```
  <procedure-identification> ::= PROCEDURE <identifier>

  <function-identification> ::= FUNCTION <identifier>
```

e.g.

```
   PROGRAM EgForward;


    PROCEDURE first(x,y,z:INTEGER);FORWARD;
      {first needs to call second}

    PROCEDURE second(i,j,k,l,m:INTEGER);
      {second needs to call first}


     BEGIN
      <statement>
      {contains a call to first}
     END;

    PROCEDURE first;


     BEGIN
      <statement>
      {contains a call to second}
     END;

   BEGIN
    <statement>
     {program block}
   END.
```

Chapter 7: Structured types

1 Enumerated, Subrange and Set types

An Enumerated type is a group of values that are named and ordered the programmer. An enumerated type is treated as an ordinal type.

A subrange type is defined as a specific subset range of any ordinal pe. Thus a subrange type can be defined as a subset range of an umerated type or as a subset range of any of the ordinal types provided QL Pascal 68000 - integer, char and Boolean (although type Boolean nstitutes only 2 values).

A set type is defined in order to represent a set or a group of values of y ordinal type. A variable of type SET represents a collection of dinal values, whereas a subrange or enumerated type variable presents one occurrence of an ordinal value.

umerated types

[illegible] type is defined by:

[illegible]

[illegible]

[illegible] is a group of programmer specified identifiers or [illegible] ordinal value associated with the identifier [illegible] correspond [illegible] position of the constant in the list starting with position 0 [illegible] type constants are like normal identifiers and are subject to [illegible] scope rules governing identifiers except that the type [illegible] variable declaration part of a block [illegible] enumerated [illegible] useful in defining a subrange type of an enumerated [illegible]

e.g.

```
TYPE Colour = (red,blue,green,orange);

     Points = (north,south,east,west);

     Letters = (a,b,d,e,c,f);
```

is an example of several valid enumerated type definitions. Operations between variables of enumerated types are governed by the assignment compatibility rules (see section 5) and enumerated type variables can act as arguments for ordinal functions:

with reference to the above example,

```
PRED(green)
```

is

```
blue
```

```
SUCC(green)
```

is

```
orange
```

```
south > north
```

yields Boolean value TRUE

```
west < east
```

yields Boolean value FALSE

```
ORD(c)
```

is 4

```
ORD(d)
```

is 2

which verifies that in Letters

d, e, c. [illegible]

The control variable in a FOR statement can be a variable of an enumerated type e.g.

```
PROGRAM enumforloop(output);

VAR

   controlvariable : (alpha,beta,gamma,delta);

BEGIN

   FOR controlvariable := alpha TO delta DO

      WRITELN(ORD(controlvariable));

   FOR controlvariable := delta DOWNTO alpha DO

      WRITELN(ORD(controlvariable))

END.
```

Subrange types

A subrange type definition is specified:

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

Subrange types can only be defined on ordinal types. The constants that delimit the range must both belong to the same host type and the first constant, known as the lower bound, must be less than or equal to the other constant, the upper bound. For other than the QL Pascal [illegible] provided ordinal types, the host ordinal type must be defined as an enumerated type. The following is an example of enumerated and subrange type definitions:

```
TYPE

  reds = (crimson, scarlet, vermillion, maroon);
                   {enumerated type}

  plus = 1..1000; {subrange type of host type integer}

  somechars = 'E'..'T';
                   {subrange type of host type char}

  somereds = scarlet..maroon;
                   {subrange type of host type reds}

  shortword = [illegible]
                   {subrange type of host type char}
```

Like enumerated type variables, operations between subrange type variables must conform to the assignment compatibility rules (see section 4). Subrange type variables can also be used as arguments for ordinal functions.

SET types

A SET type is defined as:

```
<set type> ::= [PACKED] SET OF <base type>
```

where <base type> is any identifier that conforms to normal scope rules [illegible] base type is any ordinal type. A variable of type set consists of a group of elements of type <base type> which are called members. A variable of type set can consist of any subset of the members of the base type including the full set of members and the subset containing no members [illegible]

```
TYPE

Reds = (crimson, scarlet, vermillion, maroon);
                (enumerated type)

Redset = SET OF Reds;
                (the set of the enumerated type Reds)

Acharset = SET OF 'a'..'m';
                (the set of subrange type 'a' to 'm')

RedHues : Redset; (a variable of the set type Redset)

SameHues : SET OF Reds;
                (a variable of type set of type Reds)

Afewchars : Acharset;
                (a variable of the set type Acharset)

SomeInts : SET OF [illegible]
                (a variable of type set of the [illegible]
```

[illegible]

E.g., referring to the sets of the previous example:

`[crimson] and [scarlet..maroon]`

are both valid set constructors for variables RedHues and SameHues

`['a'], ['a', [illegible] 'k'..'m']`

are both valid set constructors for variable Afewchars

`['m'..'p']`

is an invalid set constructor for variable Afewchars

NOTE Due to the large range of 32 bit Integers, 'Set of Integer' is not permissible in QL Pascal 68000.

The following relational operators are applicable to set operands:

=	test on equality
<>	test on inequality
<=	test for left hand operand being a subset of right hand operand
>=	test for left hand operand being a superset of right hand operand
IN	test for set membership

and for set variables a and b:

i) a = b yields true if all members of both a and b are identical

ii) a <> b yields true if any member of a cannot be found in b, or vice versa

iii) a <= b yields true if every member of a is also a member of b

iv) a >= b yields true if every member of b is also a member of a

v) x IN y yields true if ordinal variable x is a member of the set variable y. Here, variable x must be of the same ordinal type as the base type of the set variable y.

E.g., referring to the ongoing example:

```
[crimson, maroon] <= RedHues
```

yields true provided RedHues is constructed from members that include crimson and maroon,

and for variable Ared of type Red,

```
Ared IN RedHues
```

yields true if the shade of red assigned to Ared is a current member of RedHues.

NOTE: All relational operators applicable to sets are all at the same precedence level (see precedence rules in section 6).

Once constructed, sets can be manipulated using the following operators between set operands of the same type to yield set values of the same type as the set operands.

```
set intersection
set union
set difference
```

For two sets a and b, (a*b) is the set whose members are currently in both a and b, (a+b) is the set of members formed by merging sets a and b, (a-b) is the set of set a's members that are not also in set b.

e.g.

```
[1..6,9] * [5..7] is [5..6] [illegible]

[1..4,6] + [5,7..9] is [1..9]

[1..9] - [2..8] is [1,9]
```

Through the use of sets it is possible to produce neat, structured and comprehensible algorithmic program solutions.

PACKED data

The ISO standard specification includes the reserved word PACKED with regard to all structured data types, with the exception of pointer types, to provide the option of storing structured data contiguously, thus occupying the minimum number of media storage words required. Data can generally be packed at the expense of speed of access.

7.2 The ARRAY type

The ARRAY type is one of several structured data types provided for use in QL Pascal 68000 programs. The array is an almost universal data type among high-level programming languages.

In QL Pascal 68000 the ARRAY type defines a structure that is a uniform collection of a fixed number of components, or elements, of any simple, structured or pointer type. An array is defined:

```
<array-type> ::=
[PACKED] ARRAY
    '[' <index-type> {,<index-type>} ']'
                              OF <component-type>

<index-type> ::= <ordinal-type>
```

<index-type> can be specified as an existing type or a newly defined type. <index-type> can be defined separately or in the array definition itself. <index-type> must be an ordinal type. Thus values of type real

not be used to specify array bounds. <ordinal type> includes subrange and enumerated types. <index type> can have any number of occurrences within the square brackets in order to define what can be thought of as a multi-dimensional array.

<component type> may be of any type excluding the type of the array itself. <component type> can be an existing or newly defined type. <component type> can be defined separately or in the array definition itself. <component type> can itself be of type array. Unlike <index type>, <component type> can be of type real.

An array definition can be specified alongside an array variable declaration in the variable declaration part of a block (see section 4), e.g.

```
board : ARRAY[1..8, 1..8] OF INTEGER;
```

is a valid example of an array definition consisting of 8 rows of 8 rows of elements of type integer, a total of 64 elements. Board could also be defined:

```
board : ARRAY[1..8] OF ARRAY[1..8] OF INTEGER;
```

to achieve the same array type.

[illegible]

[illegible]

[illegible]

The following are examples of array definitions including the packed option (see PACKED data section 7.1):

```
PACKED ARRAY[1..10] OF ARRAY[1..20] OF REAL;

ARRAY[1..10] OF PACKED ARRAY[1..20] OF REAL;
```

An array variable can be referenced in its entirety, or one component at a time. Assignments may be made between array variables or between array variable components; in both cases the assignment compatibility rules apply (see section 5).

String types

String constants, or literals, may be assigned to packed array variables provided they have the same number of components as specified in the array variable definition. In such cases assignment compatibility dictates that the component type of the array is of type char, and that either, the array variable is one-dimensional or assignment is directed at one dimension of a multi-dimensional array variable. Type packed array of char is used for string types.

Accessing an array variable component is brought about by the use of indexes or subscripts which when specified in a reference to an array variable, allow immediate access to the array component through what is known as an indexed variable. For order of evaluation see appendix C. Because there is no run-time overhead when accessing array variable components, arrays, like records, are known as random-access data structures.

Indexed variable

An indexed variable is represented by:

```
<indexed-variable> =
    <array-variable> "[" <index-expression>
                     {"," <index-expression>} "]"
```

<index-expression> is an expression which is evaluated to yield a value that must be assignment compatible with the index-type of the array variable. If the value of the index expression is outside the range specified by the index type a run-time error will be generated. <indexed-variable> is a variable of the same type as the component type of the array variable and can be treated in the same way as an ordinary variable apart from acting as the control variable in a FOR statement.

When referencing an indexed variable, the number of index expressions specified must be equal to the number of dimensions of the array variable of which the indexed variable is a part.

The following is an example of a program containing array definitions and declarations.

```
PROGRAM arrays(output);
CONST
 Linelength      = 33;
 Pagelength      = 24;
 NumberOfTitles  = 2;

TYPE
 Title    = PACKED ARRAY [1..Linelength] OF CHAR;
 Titles   = ARRAY [1..NumberOfTitles] OF Title;
 Pagesize = PACKED ARRAY [1..Pagelength] OF
            PACKED ARRAY [1..Linelength] OF CHAR;

VAR
 Headings     : Titles;
 Wholepage    : Pagesize;
 Line, Column : INTEGER;
```

```
BEGIN
 Headings[3] := 'QL Pascal 68000 Reference Guide ';

 FOR Line := 1 TO Pagelength DO
   FOR Column := 1 TO Linelength DO
     Wholepage [Line,Column] :=
        Headings[3,((Line + Column - 2)
                        MOD Linelength)+1)];
 FOR Line := 1 TO PageLength DO
 BEGIN
   FOR Column := 1 TO Linelength DO
     WRITE (Wholepage[Line,Column]);
   WRITELN
 END
END.
```

PACK and UNPACK

Although occupying less space, packed data generally requires greater access time for its components; the diminution of efficiency may not warrant the space saving gained by using the packed option when defining arrays. The procedures PACK and UNPACK are specified in the ISO standard to provide for the packing and unpacking of array data.

```
PACK (<unpacked-array>,
              <starting-subscript>, <packed-array>)

UNPACK (<packed-array>,
              <unpacked-array>, <starting-subscript>)
```

PACK packs <unpacked-array> into <packed-array> starting at <unpacked-array>[<starting-subscript>]. [illegible] component of <unpacked-array> to be undefined. UNPACK unpacks <packed-array> into <unpacked-array> starting at <unpacked-array>[<starting-subscript>]. [illegible] component of <packed-array> to be undefined.

Run-time errors will occur if [illegible] sizes are inconsistent. See also appendix [illegible]

7.3.1 The RECORD type

The RECORD TYPE, like the ARRAY TYPE is another structured type that can be defined for use in QL Pascal 68000. It, too, is a structured collection of elements or components; the essential difference is that record structures are not necessarily uniform collections of components. The components of a record type are generally referred to as fields.

Record type can, if required, be specified as the component type when defining arrays.

Among other uses the Record type was included in the design of Pascal to meet the often less ordered data type requirements of the commercial world.

A record type is represented as:

```
<record-type> = [PACKED] RECORD <field-list> END[";"]
<field-list> = [((<fixed-part>[";"<variant-part>]) |
                                   <variant-part>)[";"]]
```

<field-list> is a collection of variable declaration-like data type specifications. A record type is a field list enclosed by the reserved words RECORD and END. So starting with the fixed part of a record:

```
<fixed-part> = <record-section>{";"<record-section>}
<record-section> = <identifier>{","<identifier>}
                                   ":"<type>
```

which is best expanded upon by the use of an example:

```
TYPE
PROFILE = RECORD
             ChristianName, SurName: ARRAY[1..15] OF CHAR;
             Sex: (male, female);
             Married: BOOLEAN;
             Age: 16..65
             END;
[illegible]
```

```
VAR
person:ARRAY [1..100] OF PROFILE;
employee:ARRAY[1..50] OF RECORD
                           person:profile;
                           job:worktype;
                           firstemployed:1975..1990;
                           pay:5000..30000;
                           payletter:'A'..'G'
                          END;
oneperson:profile;
secondperson:profile;
```

A record definition constitutes the region for the identifiers it contains. Within a given record definition a field identifier must be unique. The identifier does not conflict with identifiers outside its region. Thus in the example the array identifier 'person' does not conflict with the identifier 'person' of the array identifier 'employee'. This also applies to regions that contain nested record definitions; identifiers in a nested region do not conflict with identifiers local to the outer regions nested within the entire region.

Assignments may be made between variables of type record that are assignment compatible. This means that both variables must be declared using the same type. Thus in the example, oneperson, secondperson and each component of the array person are all assignment compatible:

```
oneperson := secondperson;

person[i] := oneperson;

person[i] := secondperson;
```

In such assignments, each field of the left-hand variable is assigned the value of the corresponding field of the right-hand variable.

An individual field of a record variable is referenced using a **field-designator**:

```
<field-designator> = (<record-variable>.
                      <field-specifier>)|
                      <field-designator-identifier>
```

The field designator acts as a variable identifier, with the exception of acting as a control variable in a FOR statement (see section 5).

e.g.

```
oneperson.ChristianName := 'Blaise';

secondperson.sex := Male;

person[1].Age := 33;

employee[1].person.Married := TRUE;
```

The last line above illustrates how large field designators can be for record type definitions containing structured types. In such cases the specification of record variable field access can be shortened, with the help of the WITH statement, by using a field designator identifier.

The relational operators cannot be applied to record type operands. Record variables can only be compared on a field by field basis, which can involve the use of IF statements nested to a considerable degree.

7.3.2 WITH statement

The form of the WITH statement is

```
<with-statement> = WITH <record-variable>
                   {,<record-variable>} DO
                        <statement>
```

The field identifiers of <record-variable> constitute field designator identifiers. Within <statement> either a field designator or a field designator identifier can be used to specify a record variable field access. The list of record-variables is the defining point for the field designator identifiers whose region is <statement>. <statement> is any simple or compound or structured statement.

```
WITH oneperson DO
   Age := 34;

WITH oneperson DO
 BEGIN
   Age := 35;
   ChristianName := secondperson.ChristianName;
 END;

WITH secondperson DO
 IF secondperson.Married THEN
   Age := 36;
```

are all examples of valid WITH statements.

The statement body of a WITH statement can be or can contain a WITH statement; specifying more than one record variable in the WITH statement line itself can be regarded as a nested WITH statement construct.

e.g.

```
V1,V2,...Vn are record variables

WITH V1,V2,....Vn DO <statement>
```

is equivalent to

```
WITH V1 DO
  WITH V2 DO



      WITH Vn DO <statement>
```

Conflict between identical field designator identifiers in such cases is resolved by associating the field designator identifier with the relevant record variable of the nearest WITH statement that contains the reference to the field designator identifier.

e.g.

```
WITH oneperson,secondperson DO
  Age := 36;
```

which is the same as

```
WITH oneperson DO
  WITH secondperson DO
    Age := 36;
```

means

```
WITH oneperson,secondperson DO
  secondperson.Age := 36;
```

The record variable referred to in a WITH statement is accessed before execution of the WITH statement body commences.

Variant record parts

The record type provides for the definition of versatile data structures by allowing groupings of all other data types - **type unions**. By specifying a variant record part in a record type definition, a high degree of flexibility can be introduced to such data structures. A variant record part allows for variables of different data types to be overlaid by the use of coincident selectable groupings of data type definitions. Selection of a particular grouping of data type definitions is actioned through the use of a **tag field** defined using a tag type, or just a **tag type**. The tag type must be a predefined ordinal data type. The tag field is optional; the tag type must always be present. This scheme allows for the same actual data to be associated with several variables possessing different data type definitions.

NOTE This opens up many possibilities in respect of, say, data conversion but such 'tricks' could create program portability problems, as low-level data representation is implementation dependent.

Variant part definition superficially resembles a case statement:

```
<variant-part> = CASE <variant-selector> OF
                                    <variant>{";"<variant>}

<variant-selector> = [<tag-field>":"] <tag-type>

<variant> = <case-constant-list>":" "("<field-list>")"

<case-constant-list> = <case-constant>
                                    {","<case-constant>}

<field-list> = [<identifier>{","<identifier>}
                                   ":"<type>[";"]]
```

<case-constant> must be a valid ordinal value for <tag-type> which can be any ordinal type. Each case constant within the CASE part of a variant part must be distinct and unique. The identifiers in all variant parts must be distinct and unique within the record definition although they may be re-used within nested record definitions. Field identifiers, as in the fixed part of a record definition can be defined to have any type. A field list contains zero or more identifiers. It should be noted that when the CASE construct is used with variant parts there is no corresponding

END statement

e.g.

```
TYPE

  shape = (point,circle,triangle,square);

    drawing = RECORD

                CASE figure:shape OF

                 point:();

                 circle:(radius:real);
                 triangle:
                  (side1,side2:real; angle:0..360);

                 square:(side:real)

              END;

VAR

  designpart:drawing;
```

e.g.

```
TYPE

   debtor = (credit,slpwpayer,baddebt);

   customer = RECORD

                name:ARRAY[1..30] OF CHAR;

                address:ARRAY[1..5,1..30] OF CHAR;

                CASE debtor OF
                 credit:
               (despatchdetails:ARRAY[1..20] OF CHAR);

                 slowpayer:(bankphoneno:integer);
                 baddebt:
               (liquidator:ARRAY[1..30] OF CHAR)

              END;

VAR

   accountprofile:customer;
```

drawing is an example of a record definition containing just a variant part and customer a fixed and a variant part. drawing makes use of a tag field; note that the case part of drawing is the defining point for figure. customer uses just a tag type in the case part. A tag field is interrogated to determine which grouping of fields is currently active; the currently active grouping can be changed by valid assignment to the tag field. Run time errors can occur if assignments are made in respect of groupings that are not active. Groupings that are not active are totally undefined. It is also an error to access a field with an undefined value. See appendix C. A tag field cannot be passed as a parameter in a procedure or function invocation (see section 6).

e.g. to determine which grouping is active

```
CASE designpart.figure OF
  point:<statement>;
  circle:<statement>;
  square:<statement>;
  triangle:<statement>
END;
```

or

```
WITH designpart DO
  CASE figure OF
    point:<statement>;
    circle:<statement>;
    square:<statement>;
    triangle:<statement>
  END;
```

and to change the active grouping,

```
designpart.figure := circle;
```

or

```
WITH designpart DO
  figure := circle;
```

If a tag field is not used, assignment to a field in a grouping renders that grouping active. So determining which group is active is unnecessary (and very difficult - tag type is a type definition and not a variable declaration!).

7.4.1 Pointer types

The data structure definitions dealt with so far relate to what is known as **static variables**. These are predeclared units of fixed size which exist for the entire duration of an activation of the block to which the variable is local. It is possible to create data structures which can vary in size and complexity throughout the execution of an QL Pascal 68000 program. These are known as dynamic data structures and bear no direct correlation to the static structure of an QL Pascal 68000 program. The generation and administration of dynamic data structures is handled by the predefined identifiers NEW and DISPOSE in conjunction with pointer values.

A variable of type pointer is used to reference, or indirectly access, a variable of the pointers domain type:

```
<pointer-type> = ("^" | "@") <domain-type>
```

<domain-type> is an identifier defined at a higher block level or anywhere in the same type definition part of which the pointer type identifier is part.

e.g.

```
TYPE
  portionstart = ^portion
  portion = RECORD
              order:integer;
              size:REAL;
              content:ARRAY[1..10] OF CHAR;
              colour:(red,blue,green)
            END;
  Integerpointer = ^INTEGER;
  item = ^chain;
  chain = RECORD
```

```
          chainelements:ARRAY[1..5] OF INTEGER;
          NextItemInChain:item
        END;

VAR
  longchain:item;
  oneItem:chain;
  piece:portion;
  locationofpiece:portionstart;
```

are examples of valid pointer type definitions and declarations. It is also possible to define types such as

```
TYPE
   T1 = ARRAY[1..100] OF ^T1;
   T2 = ^T2;
   T3 = record
         numero:INTEGER;
         thisrecord:^T3
        END;
```

which, though legal, are somewhat difficult to use efficiently.

Pointer variables

A variable of type pointer can be initialized or modified in one of three ways:

(i) It can be assigned the null value which is denoted by the reserved word NIL.

(ii) It can be given a unique identifying value, which serves as the address of a variable of the pointer's domain type.

(iii) It can be assigned the value of another pointer variable, acquiring the identifying value of that pointer, which may be NIL.

The reserved word NIL represents a null value unique to pointer types. It is not available for general purpose inspection. When specified for assignment or comparison with a pointer variable, the token NIL will

assume the nil-value appropriate to the pointer variable. A pointer variable to which NIL is assigned (a nil-pointer) does not reference a variable. A pointer variable can be compared to NIL or to another pointer variable with the same type. Such comparisons can only be made for equality or inequality (the relational operators = or <>) e.g.

```
PROGRAM PointerSyntax(output);

TYPE
  num1 = REAL;
  num2 = RECORD
           Int1 : INTEGER;
           Int2 : INTEGER
         END;
  ptype1 = ^num1;
  ptype2 = ^num2;

VAR
  pointer1 : ptype1;
  pointer2 : ptype2;
  pointer3 : ptype2;

 BEGIN
  pointer1 := NIL;
  pointer2 := NIL;
  pointer3 := pointer2;
  IF (pointer2 <> pointer3) OR (pointer1 = NIL) THEN
    WRITELN('pointer comparisons')

END.
```

7.4.2 NEW

The predefined procedure NEW can be invoked to dynamically allocate a new variable.

```
NEW(p)
```

creates a totally undefined variable of p's domain type, p being a variable access of any pointer type. p is said to reference this variable.

The new variable is not directly known from within an executing program and remains allocated for the duration of program execution, even if the variable allocation is initiated from within a nested subprogram block. Thus it may be necessary to reclaim the storage used by a dynamic variable and this is done by invoking the predefined procedure DISPOSE.

The full form of the procedure NEW is:

```
NEW(p,[<case-constant>{","<case-constant>}]).
```

where <case-constant> is a case constant of the variant part of a record variable access by pointer p. This form allows for more efficient storage allocation for variant records where the actual size of each record can vary depending upon which variant record grouping is currently active. The actual size required may be allocated but care must be taken to ensure that, when the storage is ready for release, the precise storage allocated is deallocated; that is, DISPOSE must be invoked using the same case constant list. It is an error if the case constant list is not identical. If a variable is created using the second form of NEW it is an error to deallocate it using the first (short) form of DISPOSE. See appendix C.

If more than one case constant is specified, then the sequence and occurrence of the case constants must correspond exactly to the full or partial variant part definition from which they are derived. It is an error if a variant that was not specified becomes active. It is an error if a variable created by the second for of NEW is accessed by the identifier variable of the variable access of a factor, of an assignment statement, or of an actual parameter. See appendix C.

7.4.3 DISPOSE

The predefined procedure DISPOSE can be invoked to de-allocate variables created by a previous invocation of NEW.

```
DISPOSE(q)
```

serves to disassociate the variable referenced by q from any pointer, q being a variable or function of any pointer type. It is an error if a subsequent attempt is made to access the variable through q, or through any other pointer, since they have become undefined. See appendix C.

It is an error to dispose of a variable that is currently being accessed or to attempt to dispose of an undefined or null-valued pointer. The full form of the procedure DISPOSE is:

```
DISPOSE(q,[<case-constant>{","<case-constant>})
```

where <case-constant> is a case constant of the variant part of a record variable access by pointer q. This form allows for more efficient storage de-allocation for variant records where the actual size of each record can vary depending upon which variant record grouping is currently active. The case constant list in DISPOSE must be identical to the case constant list in the corresponding previous invocation of NEW. It is an error if the case constant list is not identical. See appendix C. The storage released by an invocation of DISPOSE is 'given back' to the machine perhaps for re-use by further invocations of NEW.

e.g.

```
NEW(item);
          {allocate storage for a variable pointed to}

DISPOSE(item);
          {de-allocate the storage for the variable}
```

Identified variables

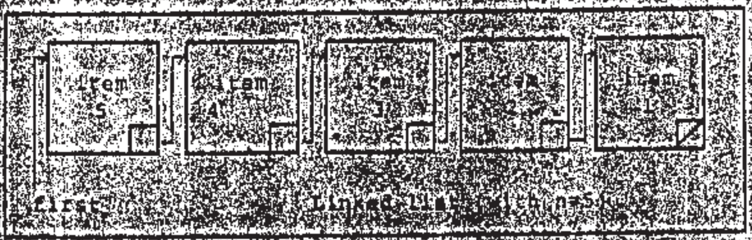
As dynamically allocated variables do not have identifiers, they are referenced through the use of identified variables.

```
<identified-variables> = <pointer-variable>("^")
```

Put succinctly, an identified variable is that which is pointed at. It is an error if <pointer-variable> is NIL or undefined. See appendex C.

Even though a function identifier may be of type pointer i.e. the function's result type, a function invocation cannot be used to construct an identified variable. It follows from a pointer type definition that an identified variable may be of any type. The following is an example of the use of pointers to establish a linked list.

Figure 2



```
PROGRAM LinkedList(output);
TYPE
   Link = ^Info;
     PieceofInfo = RECORD
[illegible]
                 ;NextInfo : Link
                 END;
VAR
     Linkdata = RECORD
[illegible]
                 END;
     F : File of Linkdata;
     Filepiece : Linkdata;
     First, InfoPointer : Link;
BEGIN
[illegible]
   First := NIL;
   FOR I := 1 TO n DO
          {n pieces of information in file F}
     BEGIN
       READ(F, Filepiece);
          {get next piece of info}

       NEW(InfoPointer);
          {allocate storage for list item}

       InfoPointer^.NextInfo := First;
          {point to previous item}
[illegible]

       First := InfoPointer;
          {store current pointer}
     END
END.
```

Typically pointers are associated with identified variables of type record, as in the previous example of the linked list.

7.5.1 File type

Apart from file types all other structured data types in QL Pascal 68000 are fixed by their definitions and declarations. File variables can be declared that are sequences of components. The size of a file variable can change during program execution and a file variable can exist outside a program. A new file may be generated by a program, or an existing file may be inspected by a program. Distinct from other structured data types (excluding dynamic data structures) files are sequential access data structures.

Files may be sequences of any data type except file types themselves or structured types that contain file type components:

```
<file-type> = FILE OF <component-type>
```

<component-type> may be an already defined data type or a new data type definition.

Space for file variables is generally allocated on rotating media devices which have long access times compared to main memory. Therefore to optimize processor throughput, main memory storage buffers are set up to contain the current 'file piece'. Such buffers may hold more than one file component and in order to access the current component, a buffer variable is provided to represent a single file component. The buffer variable is automatically allocated in conjunction with the declaration of a file variable.

```
<buffer-variable> = <file-variable>("^"|"@")
```

where <file-variable> represents a file variable access.

A buffer variable can be regarded as a 'window' that contains the current file component, through which a program can inspect a file or into which a program can generate a new component. It is an error to change the value of a file when a reference to its buffer exists.

The predefined QL Pascal 68000 type TEXT is essentially file of char with the addition of lines as an extra sequence type (see section 7.6)

7.5.2 File handling procedures

There is a number of predefined procedures and functions in QL Pascal, which relate to files in general and are detailed as follows for file f:

REWRITE(f)	This procedure statement puts f in generation mode. File f becomes empty and the buffer variable becomes undefined.
RESET(f)	This procedure statement puts f in inspection mode. After the call of reset, the buffer variable f^ represents the first component in the file. It is an error if file f is undefined before the call of RESET.
PUT(f)	This procedure statement appends the buffer variable f^ to f which must be in generation mode. It is an error if f is not in generation mode or if the buffer variable f^ is undefined or if f^ is not put on the end of the file f. After a call of put, the buffer variable becomes totally undefined. (the 'window' contents are added to the end of the file)
GET(f)	This procedure statement causes the buffer variable f^ to represent the next component in file f which must be in inspection mode. It is an error if file f is not in inspection mode. It is also an error if before the call of get there is no next component, that is, EOF(f) is true (at end of file). (the 'window' is advanced to inspect the next file component)
EOF(f)	This function call yields Boolean value true if the component represented by f^ is empty. It is an error to call EOF(f) if f is undefined.

7.5.3 READ and WRITE

These have the form:

```
READ"("[<file>","]<variable>[","<variable>]")"
WRITE"("[<file>","]<variable>[","<variable>]")"
```

<variable> is a variable declared using the same type as <file> component type. READ and WRITE can be used in place of GET and PUT, without the need to refer to file buffer variables. If <file> is not specified, READ refers to the textfile INPUT and WRITE refers to the textfile OUTPUT (see section 7.6). Note that <file> is evaluated once, regardless of the number of variables specified (See appendix C).

NOTE RESET and REWRITE have been extended to allow internal files to access named files.

```
RESET "(" <file> "," <file-name> ")"

REWRITE "(" <file> "," <file-name> ")"
```

in the case of RESET <file-name> is the name of an existing file and in the case of REWRITE <filename> is the name of a file to be created (See Appendix D).

7.6 INPUT / OUTPUT facilities

This section deals with the standard procedures that apply to textfiles.

INPUT and OUTPUT

These program parameters, which relate to the keyboard and console, are treated as textfiles. When specified, explicit definitions and declarations of these textfiles are not required, and upon program execution these input and output devices are ready for use. RESET or REWRITE must not be called for INPUT and OUTPUT.

Textfiles are sequences of char values. Text is line oriented, lines being terminated by an end of line character.

EOLN

When accessing textfile f, EOLN(f) returns TRUE if the buffer variable f^ (the current character) is the end of line character, otherwise FALSE. It is an error if f is undefined or EOF(f) is TRUE. If no file is specified, EOLN refers to file INPUT.

READLN

When accessing textfile f, READLN(f) positions the buffer variable f^ immediately after the end of line character of the current line, that is at the first character of the next line. It is an error to call READLN(f) if EOF(f) is true. If no file is specified, READLN refers to INPUT.

WRITELN

When generating textfile f, WRITELN(f) appends an end of line character to f. It is an error if f is undefined. After the WRITELN call the buffer variable f^ is undefined and f remains in the generation mode. If no file is specified, WRITELN refers to OUTPUT.

PAGE

When generating textfile f, PAGE(f) appends a page-throw character to f. If no file is specified, PAGE refers to OUTPUT. If page is used to write to a file then the effect of reading from that file is to read the formfeed character.

`General`

GET and PUT may be applied to textfiles but are cumbersome. READ, READLN, WRITE and WRITELN are almost universally applied for textfile access. Multiple arguments, as in READ and WRITE, can be specified in READLN or WRITELN calls.

Syntax:

```
READ"("[<file>","]<variable>{","<variable>}")"

READLN["("[<file>","]<variable>{","<variable>}")"]


WRITE"("[<file>","]<write-parameter>
                              {","<write-parameter>}")"

WRITELN["("[<file>","]<write-parameter>
                              {","<write-parameter>}")"]
```

<variable> can be of type real, integer or char. Thus READ and READLN will read numeric literals (see section 3.2) as a sequence of characters starting with the first non-blank character and ending with the first non-digit. If valid, the character sequence is converted to the relevant numeric type of <variable>. It is an error if the character sequence starts with a character not consistent with a numeric literal.

<write-parameter> is an expression which can incorporate formatting details. (See WRITE and WRITELN output formatting in Appendix E.)

Appendix A

Pascal syntax quick reference guide

A Pascal program has the following basic outline:

```
{program heading}
PROGRAM <heading>

{GOTO label declarations}
LABEL  1,9999;

{constant definitions}
CONST  <identifier> = <literal>;

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

{variable declarations}
VAR    <identifier(s)> : <type>;

{subprogram declarations}
PROCEDURE or FUNCTION <heading>;

BEGIN

{program statements}


END.
```

Type definitions

Predefined types:

```
INTEGER, BOOLEAN, CHAR, REAL
```

Enumerated types:

```
TYPE colours = (red, blue, green, yellow);
```

Subrange types:

```
TYPE SomeIntegers = 10..100;
     SomeColours = red..green;
```

Set types:

```
TYPE NumberSet = SET OF [illegible]
     ColourSet = SET OF SomeColours;
```

Array types:

```
TYPE AnArray = ARRAY [illegible] OF red..green;
     Paintbox = PACKED ARRAY [colours] OF BOOLEAN;
```

Record types:

```
TYPE ARecord = RECORD
               {There are 4 fixed fields}
               Field1 : INTEGER;
               Field2 : [illegible]
               Field3 : (white, grey, black);
                 RECORD
                 [illegible]
                 END;
               Field4 : ARRAY [illegible] OF [illegible]
               {and one variant field}
               CASE ATag : ATagType OF
                 Select1 : (Field5 : REAL);
                 Select2 : (Field6 : BOOLEAN);
             END;
```

File types:

```
TYPE Collection = FILE OF ARecord;
     SomeNums   = FILE OF INTEGER;
```

Pointer types:

```
TYPE Location = ^ARecord;
```

Variable declarations

```
VAR i,num,digits : INTEGER;

    SomeInfo : ARecord;
```

Procedure and Function declarations

As for the program block, except for the heading and ending with a

```
PROCEDURE ASubroutine ( i : INTEGER; VAR n : REAL );
VAR j,k : INTEGER;
BEGIN
 .
 .                                   {procedure statements}
 .
END;

FUNCTION ASubroutine : REAL;
VAR i,j,k : INTEGER;
BEGIN
 .
 .                                   {function statements}
 .
ASubroutine := 5.0
END;
```

Statements

Assignment statements:

```
Answer := Result;

Answer := a * b / c + d;

ASet := [1, 2, 3, x..y, 7];
```

Goto statements:

```
GOTO 2;

2 : x:= y; {target}
```

If statements:

```
IF (Answer = 5) OR (Result <> 7) THEN
  BEGIN
    .
    .                              {statements}
    .
  END
ELSE
  BEGIN
    .
    .                              {statements}
    .
  END;
```

For statements:

```
FOR i := 10 TO 20 DO {(or FOR i := 20 DOWNTO 10 DO)
  BEGIN
    .
    .                                 {statements}
    .
  END;
```

While statements:

```
WHILE NOT (Answer >= 5) AND (RESULT < 12) DO
  BEGIN
    .
    .                                 {statements}
    .
  END;
```

Repeat statements:

```
REPEAT
    .
    .                                 {statements}
    .
UNTIL (Answer [illegible] OR (RESULT [illegible]
```

Case statements:

```
CASE Answer OF
 1,2 : BEGIN
         .
         .
         .
       END;
 5 : <statement>
END;
```

With statements:

```
WITH ARecord DO
  BEGIN
    Field := 5;
    .
    .
    .
  END;
```

Arithmetic expressions:

```
  Num1 + Num2
  Num1 - Num2
  Num1 * Num2
  Num1 / Num2
  Num1 DIV Num2                  [illegible]
  Num1 MOD Num2
```

Appendix B: Compile-time error messages

1 : Illegal character
2 : Illegal character
3 : File ends inside quoted string
4 : File ends inside a comment
5 : Integer part of number is too large
6 : PROGRAM expected
7 : Identifier expected
8 : ')' expected
9 : ';' expected
10 : A block cannot start with this symbol
11 : Missing dot at end of program
12 : Text encountered after end of program
13 : BEGIN expected
14 : A procedure has been declared as forward but has not been found
15 : Syntax error
16 : A label must be an INTEGER constant
17 : Label number expected
18 : '=' expected
19 : Type has been implicitly declared, but actual definition not found
20 : ',' expected
21 : Undeclared label
22 : This kind of identifier cannot be used to start a statement
23 : Type expected
24 : 'OF' expected
25 : '(' expected
26 : Line too long, it will be truncated
27 : Only two digits are permitted in the E field of a real number
28 : Unexpected end of source file encountered
29 : Commas must be used between labels
30 : A type identifier must follow '^'
31 : '[' expected
32 : ']' expected
33 : Files cannot contain files
34 : END expected
35 : '..' expected

36 Type mismatch between subrange bounds
37 The first bound of the subrange is greater than the second
38 Illegal subrange type
39 Constant expected
40 Number expected
41 Type identifier expected
42 Identifier already declared in this block
43 Identifier not declared
44 Too many elements in type
45 Type is not countable
46 Constant must be of another type
47 Block name expected
48 The previous forward declaration does not agree
49 The parameter list should not be repeated
50 This block has been declared as forward for the second time
51 Parameter expected
52 Function return type must be pointer, subrange, real or ordin
53 Maximum code size for main procedure exceeded
54 * expected
55 Cannot READ or WRITE zero items
56 A field width must be of type integer
57 Expression cannot be written
58 The = and <> operators cannot be used between these typ
59 An expression of type PACKED ARRAY OF CHAR required
60 The IN operator cannot be used between these types
61 The + and - operators can only be used on integer and re types
62 The OR and AND operators can only be used between boole operands
63 The + and - operators cannot be used between these operan
64 [illegible]
65 Unimplemented feature
66 The MOD and DIV operators may only be used between integ operands
67 The * operator may not be used between these operands
68 Invalid operand
69 The NOT operator can only be applied to boolean operands
70 The ^ symbol may only be used for pointer and file variables
71 Internal compiler error
72 A dot follows a variable which is not a record
73 Field not known

74 Only arrays may be subscripted
75 The expression type is incompatible with the index type of th array
76 := expected
77 Variable and expression are not assignment compatible
78 Expression too complex
79 DO expected
80 UNTIL expected
81 THEN expected
82 The variable of a for loop must be a local variable
83 The variable of a for loop must be of an ordinal type
84 TO or DOWNTO expected
85 Subscript value out of bounds
86 Division by zero
87 Case label expected
88 Empty case statement body
89 The case constant appears twice
90 Parameter list expected
91 Number of parameters does not agree with declaration
92 Extra comma; it will be ignored
93 Variable of different type required
94 An element of a packed structure cannot be used as a parameter
95 Procedural parameter is not identical to the requirements the parameter list
96 Expression of different type required
97 The argument to NEW or DISPOSE must be a pointer
98 Only the current function may be assigned to
99 A boolean expression is required
100 The empty string is not permitted
101 Label already defined
102 Label has been declared but not defined
103 Label already declared
104 Placement of label invalidates previous GOTO statement
105 Label numbers must be in the range 0 to 9999
106 Label is not accessible from this point in the program
107 The identifier cannot be redefined in this scope
108 External procedures may only be declared at the outermo level
109 RESET and REWRITE may only be applied to files
110 RESET and REWRITE may not be used on the standard file

input and output

111 : READLN, WRITELN and PAGE may only be applied to textfiles

112 : Cannot write to input or read from output

113 : Record type required

114 : A file is required here

115 : Items within a set constructor must have identical types

116 : Not enough space, try increasing workspace size

117 : The MOD operator must have a positive, non zero, argument

118 : Unimplemented instruction

119 : Parameter should be of type unpacked array

120 : Parameter should be of type PACKED array

121 : Subscript parameter is incompatible with the subrange of the unpacked array parameter

122 : Array host types are not identical

123 : Same control variable in nested for statements

124 : Cannot assign to a for statement control variable

125 : Cannot pass a for statement control variable as a variable parameter to a subprogram

126 : Cannot call READ or READLN with a for statement control variable as parameter

127 : For statement control variable is threatened by a procedure or function

128 : The argument to DISPOSE must be a variable or function of type pointer

129 : The argument to INCLUDE must be a filename in quotes

130 : Unable to open INCLUDE file for input

131 : INCLUDE cannot be nested to this depth

132 : Too many case constants supplied

133 : Case constants can not be variables

134 : This case constant does not match any of the variants

135 : This case constant is type incompatible with the corresponding variant

136 : A string can not be on more than one line

137 : The / operator may not be used between operands of these types

138 : The left hand argument of the IN operator must be ordinal

139 : File variables or structured variables with file components cannot be value parameters

140 : The case index must be an expression of ordinal type

141 : Field width must be an expression of ordinal type

142 : This function does not contain an assignment to its identifier

143 : Files and structured types containing files can not be assigned

144 : The actual parameter corresponding to a variable parameter must be a variable access

145 : A pointer variable must be a variable access

146 : The case constant list is incomplete

147 : This parameter cannot denote a field that is the selector of a records variant part

148 : The applied occurrence of the type identifier is within the scope of the field designator of the same name

149 : This case constant can never be reached

150 : Only integer, real or character values can be read from a textfile

151 : Variables in set constructors must be in the range 0..255

152 : Possible unclosed comment

153 : Program parameters can only be defined as variables

154 : Drive full

Appendix C: Collected errors

The following is a list of collected errors. They are all trapped by the QL Pascal run-time system with the exception of those marked by a asterisk (*). These errors mainly involve undefined variables or dynami storage.

Array Types and Packing

1. It is an error if the value of any subscript of an indexed-variable i not assignment-compatible with its corresponding index-type.

2. In a call of the form PACK (Vunpacked, StartingSubscrip Vpacked), it is an error if the ordinal-typed actual paramete (StartingSubscript) is not assignment-compatible with th index-type of the unpacked array parameter (Vunpacked).

3. In a call of the form PACK (Vunpacked, StartingSubscrip Vpacked), it is an error to access any undefined component c Vunpacked.

4. In a call of the form PACK (Vunpacked, StartingSubscrip Vpacked), it is an error to exceed the index-type of Vunpacked.

5. In a call of the form UNPACK (Vpacked, Vunpackec StartingSubscript), it is an error if the ordinal-typed actua parameter (StartingSubscript) is not assignment-compatible wit the index-type of the unpacked array parameter (Vunpacked).

6. In a call of the form UNPACK (Vpacked, Vunpackec StartingSubscript), it is an error for any component of Vpacked t be undefined.

7. In a call of the form UNPACK (Vpacked, Vunpackec StartingSubscript), it is an error to exceed the index-type c Vunpacked.

Record Types

[illegible] It is an error to access or reference any component of a record variant that is not active.

9. It is an error if any constant of the tag-type of a variant-part does not appear in a case-constant-list.

10. It is an error to pass the tag-field of a variant-part as the argument of a variable-parameter.

11* It is an error if a record that has been dynamically allocated through a call of the form NEW (p,C1..,Cn) is accessed by the identified-variable of the variable-access of a factor, of an assignment statement, or of an actual parameter.

File Types, Input and Output

12* It is an error to change the value of a file variable f when a reference to it's buffer, buffer variable f, exists.

13. It is an error if, immediately prior to a call of PUT, WRITE, WRITELN or PAGE, the file affected is not in the generation state.

14. It is an error if, immediately prior to a call of PUT, WRITE, WRITELN or PAGE, the file affected is undefined.

15. It is an error if, immediately prior to a call of PUT, WRITE, WRITELN or PAGE, the file affected is not at end of file.

16. It is an error if the buffer variable is undefined immediately prior to the use of PUT.

17. It is an error if the affected file is undefined immediately prior to any use of RESET.

18. It is an error if, immediately prior to use of GET or READ, the file affected is not in the inspection state.

19. It is an error if, immediately prior to use of GET or READ, the file affected is undefined.

20. It is an error if, immediately prior to use of GET or READ, the file affected is at end-of-file.

21. It is an error if, in a call of READ, the type of the variable-access is not assignment compatible with the type of the value READ (and represented by the affected file's buffer-variable)

22. It is an error if, in a call of WRITE, the type of the expression is not assignment compatible with the type of the affected file's buffer-variable.

23. In a call of the form EOF(f), it is an error for f to be undefined.

24. In any call of the form EOLN(f), it is an error for f to be undefined.

25. In any call of the form EOLN(f), it is an error for EOF(f) to be true.

26. When reading an integer from a textfile, it is an error if the input sequence (after any leading blanks or end-of-lines are skipped) does not form a signed-integer.

27. When an integer is read from a textfile, it is an error if it is not assignment compatible with the variable-access it is being attributed to.

28. When reading a number from a textfile, it is an error if the input sequence (after any leading blanks or end-of-lines are skipped) does not form a signed-number.

29. It is an error if the appropriate buffer variable is undefined immediately prior to any use of READ.

30. In writing to a textfile, it is an error if the value TotalWidth or FractionalDigits, if used, is less than one.

Pointer Types

31. It is an error to try to access a variable through a NIL valued pointer.

32. It is an error to try to access a variable through an undefined pointer.

Dynamic Allocation

33. It is an error to try to dispose of a dynamically allocated variable when a reference to it exists.

34. When a record with a variant part is dynamically allocated through a call of the form NEW(p,C1,...,Cn) it is an error to activate a variant that was not specified unless it is at a deeper level than Cn.

35. It is an error to use the short form of DISPOSE, i.e. DISPOSE(p) to deallocate a variable that was allocated using the long form i.e. NEW(p,C1,...,Cn).

36. When a record with a variant part is dynamically allocated through a call of the form NEW(p,C1,...,Cn) it is an error to specify a different number of variants in a call of DISPOSE.

37. When a record with a variant part is dynamically allocated through a call of the form NEW(p,C1,...,Cn) it is an error to specify a different number of variants in a call of DISPOSE.

38. It is an error to call DISPOSE with a NIL-valued pointer argument.

39. It is an error to call DISPOSE with an undefined pointer argument.

Required Functions and Arithmetic

40. For a call of the SQR function, it is an error if the result does not exist.

41. In a call of the form LN(x) it is an error for x to be less than or equal to zero.

42. In a call of the form SQRT(x) it is an error for x to be negative.

43. For a call of the function TRUNC it is an error if the result is not in the range -MAXINT..MAXINT.

44. For a call of the function ROUND it is an error if the result is not in the range -MAXINT..MAXINT.

45. For a call of the function CHR, it is an error if the result does not exist.

46. For a call of the function SUCC, it is an error if the result does not exist.

47. For a call of the function PRED, it is an error if the result does not exist.

48. In a term of the form x/y it is an error for y to equal zero.

49. In a term of the form i DIV j it is an error for j to equal zero.

50. In a term of the form i MOD j it is an error if j is zero or negative.

51. It is an error if any integer arithmetic operation, or function whose result type is integer, is not computed according to the mathematical rules for integer arithmetic.

Parameters

52. It is an error if an ordinal-typed value-parameter and it's actual-parameter are not assignment compatible.

53. It is an error if a set-typed value-parameter and it's actual-parameter are not assignment compatible.

Miscellaneous

54*. It is an error for a variable-access contained by an expression to be undefined.

55*. It is an error for the result of a function call to be undefined.

56. It is an error if a value and the ordinal-typed variable, or function-designator it is assigned to, are not assignment-compatible.

57. It is an error if a set-typed variable, and the value assigned to it, are not assignment compatible.

58. On entry to a case-statement, it is an error if the value of the case-index does not appear in a case-constant-list.

59. If a for-statement is executed, it is an error if the types of the control-variable and the initial-value are not assignment-compatible.

60. If a for-statement is executed, it is an error if the types of the control-variable and the final-value are not asssignment-compatible.

Order of Evaluation:

The order of evaluation of

a. the indices of multidimensional arrays

b. the constituent members of set-contructors

c. member-designators in set-contructors

d. actual parameters in function and procedure calls

e. either side of assignment statements

f. the parameters of PACK and UNPACK

is generally left to right although the order may depend upon optimisation features of the compiler.

In Boolean expressions not all of the operands may need to be evaluated. Thus if operands have side-effects (eg function calls) the results may not be predictable.

Appendix D

Extensions to the ISO Standard

The following extensions to the ISO standard can only be used wh EXTEND is specified as a compile time option [illegible] QL Pascal.

RESET and REWRITE

These two predefined procedures have been extended to allo internal files to access named files.

```
RESET (<file>, <file-name>)
REWRITE (<file>, <file-name>)
```

In the case of RESET, <file-name> is the name of an existing file and in the case of REWRITE, <file-name> is the name of the file to create.

<file-name> is the name of the file as understood by the operating system. It may be specified as a string literal or by using a variable of type packed array of char containing the <file-name>. Th string (or the array) may contain leading and/or trailing spaces whi will be ignored.

Printing from a Pascal program

The extended form of REWRITE can be used to perform output to printer connected to the serial line SER1 for example.

```
PROGRAM printeg;

  TYPE
      pfile = TEXT;

  VAR  printer : pfile;
```

```
BEGIN

  REWRITE(printer, 'SER1')

  WRITELN(printer, 'Hello')

END
```

INCLUDE

This predefined directive allows additional program fragments to be included in the source program at compile time. The format is

```
INCLUDE <file-name>
```

<file-name> is the name of the file as understood by the local operating system and must be specified using a string literal. An error will occur if the include file cannot be opened. INCLCDE may be nested to a depth of three and it is an error to exceed this.

EXTERNAL

This directive can be specified in the declaration of a function or procedure in the main program. It allows a subprogram to be declared as external and to be defined elsewhere. The subprogram identifier and its formal parameter list (and the result type if it is a function) are specified followed by the reserved word EXTERNAL and a unique number.

```
PROGRAM Extern[illegible]
```

```
PROCEDURE ext1 (x,y,z : INTEGER); EXTERNAL 175;
.
.
.
FUNCTION ext2 (a : BOOLEAN) : REAL ; EXTERNAL 176;
.
.
.
BEGIN

   {Calls to procedure ext1 and function ext2 are
valid anywhere within the main program block.}

END.
```

This extension allows users with Metacomco's BCPL compiler (or assembler) to write BCPL (or BCPL look-alike) programs which may be linked with a Pascal program.

Linking an Assembler module to a Pascal program

Essentially the assembler has to look like a BCPL module. The format of this is:

```
Length of module in longwords (1 longword)

code
 .
 .
 .
code

(The following global information must be longword
 aligned)

global number  (1 longword)
program offset (1 longword)
global number  (1 longword)
program offset (1 longword)
```

```
[illegible] global number [illegible]
```

This defines all routines that you have used in your assembler module [illegible] global.

Your Pascal program also has to be told this information. This is done by declaring the relevant procedure or function as EXTERNAL followed by the appropriate global number.

Global numbers should be in the range [illegible] since lower numbers are used by the run-time system.

[illegible] you are [illegible] assembler code [illegible] your own, you will need to use [illegible] the relevant [illegible] from putting additional information on the end of the module to be linked. See your [illegible] Development Kit manual and the example in Appendix [illegible] for more details.

QTRAP

This predefined procedure allows QDOS traps to be called from Pascal. The routine has the following format:

```
QTRAP (<trap number>, <[illegible]>);
```

The first argument is the trap number to the QDOS: a number 1, 2 or 3 and must be of type integer. The second argument is a structure which allows the data registers D0, D1, D2 and D3 and the address registers A0, A1, A2 and A3 to be set up with the relevant call parameters. This argument must be of type record as follows:

```
RECORD
   D0 : INTEGER;
   D1 : [illegible]type;
   D2 : [illegible]type;
   D3 : [illegible]type;
```

```
   A0 : [illegible]type;
   A1 : [illegible]type;
   A2 : [illegible]type;
   A3 : [illegible]type;
END;
```

The required QDOS trap function is selected by field D0. This field must be of type integer. The other fields should be of types compatible with the data to be passed to the trap. The fields corresponding to registers that are not used by the trap should be of type integer.

[illegible] information is returned [illegible] the third argument [illegible] structure [illegible] is similar [illegible] field should be compatible [illegible] QDOS error return is passed back [illegible] field [illegible].

CHANNELID

Many of the QDOS traps [illegible] with the QTRAP procedure require the channel ID to be specified, typically in address register A0. This procedure allows easy access to this information. The format is as follows:

```
CHANNELID (<file>, <channel>);
```

The first argument must be a previously defined file variable and the second argument must be a variable of type integer. After the call this variable contains the channel ID if the file was open or zero otherwise.

The Graphics Include File

On the supplied microdrive cartridge 'B' you will find a file called 'graphics_INC'. This is an include file containing a set of external function and procedure declarations which allow many useful routines to be called from Pascal. To use this file it should be included in your main program as follows:

```
INCLUDE mdv1_graphics_INC
```

Note that if you are using an unexpanded (128K) QL then you may find it necessary to reduce the code in the include file to just those routines that will be called.

The routines in this file are:

i) Random

```
FUNCTION random
(seed : INTEGER) : INTEGER;
```

This routine returns the next psuedo random number from a sequence identified by the argument seed. If the result of the previous call to random is used as the seed for the next call, the sequence will not repeat until all possible numbers have been generated.

ii) Time

```
FUNCTION time : INTEGER;
```

When a Pascal program is started the current value of the clock is stored. A call to time will return the difference between the new current time and the initial time. The result is in seconds.

iii) Timeofday

```
PROCEDURE timeofday
(VAR hh, mm, ss : INTEGER);
```

This procedure returns the current time (assuming that this has been set correctly when the machine was first started). The time is returned in the three integer arguments passed to the procedure.

iv) Strtimeofday

```
PROCEDURE strtimeofday
(VAR h1,h2,colon1,m1,m2,colon2,s1,s2 :
 CHAR);
```

This procedure returns the current time (assuming that this has been set correctly when the machine was first started). The time is returned in the eight character arguments passed to the procedure. The hour is passed back in the first and second arguments, the minutes in the fourth and fifth and the seconds in the seventh and eighth. The third and sixth arguments are passed back as colons for convenience

v) Date

```
PROCEDURE date
(VAR year, month, day : INTEGER);
```

This procedure returns the current date (assuming that this has been set correctly when the machine was first started). The numeric date is returned in the three integer arguments passed to the procedure.

vi) Strdate

```
PROCEDURE strdate
(VAR y1,y2,y3,y4,space1,m1,m2,m3,space2,d1,d2
 : CHAR);
```

This procedure returns the current date (assuming that this has been set correctly when the machine was first started). The date is returned in the eleven character arguments passed to the procedure. The year is passed back in the first four arguments, the month in the sixth, seventh and eighth and the date in the tenth and eleventh. The fifth and ninth arguments are passed back as spaces for convenience

[illegible] Screen

```
FUNCTION screen1
(code1: INTEGER) [illegible]

FUNCTION screen2
(code2, arg1: INTEGER) [illegible]

FUNCTION screen3
(code3, arg1, arg2: INTEGER) [illegible]
```

The screen functions are generalised [illegible] for handling the QL screen. The exact operation is determined by the [illegible] operand [illegible] require [illegible] number of the screen function refers to the number of parameters it requires. All three functions return an error code which will be zero if all is well and a negative QL [illegible]

The codes have been given [illegible] declared in the supplied include file [illegible] codes [illegible]

```
screen3 (ScreenBorder, [illegible])
```

Set window border to the specified colour and width. The border runs inside the window limits and is doubled in the vertical edges.

```
screen1 (ScreenCursor)
```

Enable the cursor. It is automatically enabled when a buffered read from the screen is pending. Without an enabled cursor in a window CTRL-C cannot be used to switch to the new job even if an unbuffered read is in operation.

```
screen1 (ScreenNocursor)
```

Disable the cursor.

```
screen3 (ScreenAt, column, row)
```

Position the cursor at the specified row and column using character coordinates.

```
screen3 (ScreenAtp, x, y)
```

Position the cursor at the specified point, using pixel coordinates. The position refers to the top left corner of the next character rectangle relative to the top left corner of the window.

```
screen2 (ScreenTab, column)
```

Tab to column specified.

```
screen1 (ScreenNewline)
screen1 (ScreenLeft)
screen1 (ScreenRight)
screen1 (ScreenUp)
screen1 (ScreenDown)
```

Move the cursor to the start of the next line or one space in the relevant direction.

```
screen2 (ScreenScroll, dist)
screen2 (ScreenScrolltop, dist)
screen2 (ScreenScrollbottom, dist)
```

Scroll all the screen, that part above the cursor line or that part below the cursor line the specified distance in pixels. A positive value for dist will move the screen down while a negative distance scrolls it up. Blank space is filled with the current paper colour.

```
screen2 (ScreenPan, dist)
screen2 (ScreenPanline, dist)
screen2 (ScreenPanEOL, dist)
```

Pan all of the screen, the current cursor line or the right hand end of the cursor line the specified distance in pixels. The right hand end starts at the current cursor column. A positive value for dist will move the lines to the right while a negative value moves it to the left. Blank space is filled with the current paper colour.

```
screen1 (ScreenClear)
screen1 (ScreenCleartop)
screen1 (ScreenClearbottom)
```

```
screen1 (ScreenClearLine)
screen1 (ScreenClearEol)
```

Clear the screen, or part of it, to the current paper colour. Part screens are defined as in [illegible] above.

```
screen2 (ScreenPaper, colour)
screen2 (ScreenStrip, colour)
screen2 (ScreenInk, colour)
```

Set the paper, strip or ink to the specified colour.

```
screen2 (ScreenFlash, switch)
screen2 (ScreenUnderline, switch)
screen2 (ScreenFill, switch)
```

Sets flashing, underlining or screen fill mode on or off. If switch is 0 then it is turned off, if it is 1 then it is turned on.

```
screen2 (ScreenMode, mode)
```

Sets the screen printing mode. If mode is -1 then ink is exclusive ORed into the background. If mode is 0 the character background is the current strip colour, and if it is 1 then the background is transparent. For the latter two values plotting will be done in the current ink colour.

```
screen3 (ScreenSize, width, height)
```

Sets the size of characters. Width is a number in the range 0 to 3 and indicates widths of 6, 8, 12 or 16 pixels. Height is 0 for 10 pixels and 1 for 20 pixels. In 8 colour mode only 12 or 16 pixel widths are allowed.

viii) Window

```
FUNCTION window5
(code : INTEGER; VAR w, h, x, y : INTEGER) :
INTEGER;
```

```
FUNCTION window6
(code : INTEGER; VAR w, h, x, y : INTEGER;
 colour : INTEGER) : INTEGER;

FUNCTION window7
(code : INTEGER; VAR w, h, x, y : INTEGER;
 colour : INTEGER; width : INTEGER) : INTEGER;
```

The window functions are general purpose routines for manipulating windows.

The first argument, code, describes the action to be taken. As with the screen functions, these codes are have been given suitable Pascal constants defined in the supplied include file 'mdv1_graphics_codes_INC'.

The next four arguments are used to specify the window; a width w and a height, h followed by the x coordinate and the y coordinate (x being measured to the right and y down from some origin.) Where appropriate, the next two arguments represent a new colour and border width, respectively. Colour is used when defining a new window or filling a block within a window.

```
window5 (WindowAskp, w, h, x, y)

window5 (WindowAskc, w, h, x, y)
```

Return the size of the window in w and h and the cursor position relative to the top left corner in x and y. window.askp returns the information in pixel coordinates, window.askc returns it in character coordinates.

```
window7 (WindowDefine, w, h, x, y, colour, width)
```

Define a new window as specified by w, h, x, y. The size is given in pixels in w, h and the position, also in pixels, in the x, y refers to the top left corner of the window relative to the top left of the screen. Width and colour define the border width (in pixels) and border colour, respectively.

```
window7 (WindowFillblock, w, h, x, y, colour)
```

Fill a block in a window. The size of the block is given in pixels by w and h. The top left corner of the block is given in pixels by x and y. The last argument colour is only relevant for the call of window. The block is filled with the specified colour (or stipple) according to the current overprinting mode.

```
[illegible]

FUNCTION plot1
(code : INTEGER; x [illegible]) : INTEGER;
FUNCTION plot2
(code : INTEGER; [illegible]
[illegible]
FUNCTION plot3
(code : INTEGER; [illegible]
[illegible]
FUNCTION plot6
(code : INTEGER; [illegible]
[illegible] : INTEGER;
```

These are generalised graphics routines for plotting lines and arcs in the [illegible]. They take a code value and up to five real arguments. The first argument, code, describes the action to be taken. As with the screen [illegible], these codes have been given suitable Pascal constants [illegible] the supplied include file [illegible] graphics codes [illegible]

`plot3 (PlotPoint, x, y)`

Plot a point at position x, y.

`plot5 (PlotLine, xs, ys, xf, yf)`

Plot a line starting at xs, ys and finishing at xf, yf.

`plot6 (PlotArc, xs, ys, xf, yf, angle)`

Plot an arc starting at xs, ys and finishing at xf, yf. The value of angle indicates the angle subtended by the arc.

`plot6 (PlotEllipse, x, y, [illegible] angle)`

Plot an ellipse centered at x, y with [illegible] The value of angle indicates the rotation angle.

`plot4 (PlotScale, [illegible])`

Set the origin at x, y with length of [illegible]

`plot3 (PlotCursor, [illegible])`

[illegible] the graphics cursor [illegible] the current graphics point.

recolour

```
FUNCTION recolour
(ch : [illegible]; c0, c1, c2, c3, c4, c5, c6, c7 : INTEGER) :
  INTEGER;
```

The window has each of the colours replaced by an alternative. The eight arguments are the colour numbers in the range 0–7 representing the new colour required.

Appendix E:

WRITE and WRITELN OUTPUT Formatting

Each expression to be output can have a **Totalwidth** field associated with it:

```
<write-parameter> = <expression>[":"<totalwidth>]
```

<totalwidth> is an expression that represents a positive integer amount and is the number of spaces allocated for outputting the result of <expression>. The result is right-aligned in the field of spaces. It is an error if <totalwidth> is less than 1. If <totalwidth> is omitted default values are assumed and are as follows:

for <expression> result Boolean, <totalwidth> defaults to the value necessary to output the Boolean values FALSE or TRUE without leading spaces

for <expression> result char, <totalwidth> defaults to the value 1

for a string literal, <totalwidth> defaults to the value required to output the full string without leading spaces

for <expression> result integer, <totalwidth> defaults to value 12

for <expression> result real, <totalwidth> defaults to value 13 (the default output representation for real is that of floating-point)

`Boolean and string-literals`

If <totalwidth> is smaller than the size required to output a Boolean or string literal, then only the first <totalwidth> characters of the literal are output.

If <totalwidth> is larger than the size required to output a Boolean or string literal or a character, then the full literal is output preceded by <totalwidth> - size blanks where size is the actual size of the literal.

`Integer literals`

The sign, if negative, and all significant digits of an integer are always output: leading zeros are suppressed. If <totalwidth> is larger than the field required to fully output an integer literal, then the literal is preceded by <totalwidth> - size blanks, where size is the actual size of the full integer literal (with leading zeros suppressed). Integer zero is output as 0.

`Real number literals`

Real number literals can be output in two different ways:

```
<real> : <totalwidth>
```

```
<real> : <totalwidth> : <fractional-digits>
```

The first form outputs the literal in floating point format and the second form outputs the literal in fixed point format.

`Floating-point format`

This takes the form:

(i) a negative sign if <real> is less than zero, otherwise a blank
(ii) the first non-zero digit of <real>
(iii) a decimal point
(iv) enough digits of <real>, up to a maximum of 6 for single precision, to equal <totalwidth>
(v) 'E' followed by the sign of the exponent followed by 2 digits for the exponent itself.

As floating-point numeric literals cannot be preceded by blanks, <totalwidth> should be specified accurately, otherwise the output literal may be preceded by spurious least significant digits.

`Fixed-point format`

This takes the form:

(i) <totalwidth> - <actual characters> blanks if <totalwidth> is greater than <actual characters> where <actual characters> = <fractional digits> + <number of digits in integral portion of <real>> + 1. If <real> is less than zero, <actual characters> is further incremented by 1.

(ii) a negative sign if <real> is less than zero.

(iii) the integral portion of <real> followed by a decimal point.

(iv) <fractional digits> of the fractional portion of <real>.

Regardless of <totalwidth> a minimum of <actual characters> is always printed.

The following are examples of formatted output statements:

```
[illegible] larger [illegible]
```

produces

```
CHAC[illegible]
```

```
Writeln(12345678[illegible]
```

produces

```
12345678          90
```

```
Writeln(Areal:10); where Areal = 123.456
```

produces

```
 1.239E+02
```

Appendix F: Example Programs

Example 1; Digital Clock

```
{ This example program produces a movable clock on
  the screen. Use EXEC to run, CTRL-C to get to the
  window, cursor keys to move it where you want and
  <RETURN> to start displaying the time. Originally
  written by Alan Cosslett (MetaComCo) in BCPL,
  rewritten in PASCAL by Peter Carr (MetaComCo)
                                          May 1985 }

PROGRAM CLOCK (OUTPUT,INPUT);
CONST
  INCLUDE 'MDV2_GRAPHICS_CODES_INC';

CHARSINCLOCK       = 8;     { i.e., 'hh:mm:ss' }
HEIGHTOFCHAR       = 10;    { In pixels        }
BORDERWIDTH        = 1;     { In pixels        }
DOWN               = 216;   { Down arrow       }
UP                 = 208;   { Up arrow         }
LEFT               = 192;   { Left arrow       }
RIGHT              = 200;   { Right arrow      }
ENTER              = 10;    { Enter to display time }
ERROR              = 0;     { Can't read keyboard   }

VAR
 CHSIZE,ERR,WIDTH,HEIGHT,XCOORD,YCOORD : INTEGER;

INCLUDE 'MDV2_GRAPHICS_INC';

PROCEDURE INITIALISE;
{  This routine works out which mode we are in by
  seeing how many characters wide the default window
  is (this is 37 for TV mode and 74 for monitor mode).
  A window that is big enough to display the clock and
  a white border is then defined. }
```

external directive.

The BCPL routine is a PEEK function which returns the contents of the specified memory location; the assembler routine is POKE which allows a given memory location to be changed.

The three routines are connected by the BCPL global vector (see *BCPL Development Kit* for details).

Each routine must be compiled using the relevant compiler or assembler. They can then be linked together using PASLINK (see Foreword for details).

Note that both Peek and Poke could have been written in BCPL or assembler.

```
{************** Pascal Program ****************
*
*  An example of how easy it is to link in
*  functions written in BCPL and assembler. This
*  program must be linked with the BCPL program
*  Peek and the assembler program Poke.
*
***********************************************}

PROGRAM MEMORY ( INPUT,OUTPUT );

VAR
 ADDRESS,CONTENTS,VALUE : INTEGER;

{ External assembler routine to Poke a location }

PROCEDURE POKE(ADDRESS, VALUE : INTEGER);
          EXTERNAL 175;

{ External BCPL routine to Peek a location }

FUNCTION  PEEK(ADDRESS : INTEGER) : INTEGER;
          EXTERNAL 176;
```

```
BEGIN
WRITELN('Input address to change ');
READLN(ADDRESS);

WRITELN('Input new value');
READLN(VALUE);

POKE(ADDRESS,VALUE);

WRITELN('Long word contents of ',ADDRESS,
        ' changed to ',VALUE);

WRITELN('Input address to examine');
READLN(ADDRESS);

CONTENTS:=PEEK(ADDRESS);

WRITELN('Longword contents of ',ADDRESS,
        ' is ',CONTENTS)

END. { Memory }


//**************  BCPL Program  *****************
//                                               *
//   This routine examines an address in memory  *
//   and returns the longword contents. It must  *
//   be linked to the Pascal program Memory with *
//   the assembler program Poke.                 *
//                                               *
//***********************************************

SECTION "peek"

GET "LIBHDR"

GLOBAL $( peek : 176          // Peek is global 176
       $)
```

```
* Convert BCPL address to machine address and
* return the contents to Pascal
*
* [illegible](address) = [illegible](address >> 2)
*
*
************* Assembler Program ***************
*
* This routine updates the memory location given
* as the first argument (D1) with the value given
* as the second argument (D2). It must be linked
* to the Pascal program Memory with the BCPL
* program Peek.
*
*
***********************************************
*
* POKE is global number 175
*
G_POKE  EQU     175
*
* Section begins with a length count
*
MODS    DC.L    (MODE-MODS)/4
*
* Update address with supplied value
*
POKE    MOVE.L  D2,0(A0,D1.L)
*
* Return to Pascal
*
        JMP     (A6)
*
* Ensure global information is longword aligned
*
        CNOP    0,4
*
* Mark beginning of global/offset list
*
        DC.L    0
*
* Define global number and offset in program
```

```
*
        DC.L    G_POKE,POKE-MODS
*
* Mark highest referenced global
*
        DC.L    100
MODE    END
```

Appendix G - Compliance Statement

QL Pascal 68000 is an implementation of a standard Pascal which has passed validation by the British Standards Institution under the ISO Standard 7185 "Specification for computer language PASCAL". The implementation defined features are as follows:

E.1 The value of each char-type corresponding to each allowed string character is the corresponding ISO character. See ISO 646 (ASCII).

E.2 The subset of real numbers denoted by signed real are the values representable with 32 bit floating point. This is about 7 decimal places.

E.3 The values of char type are the ISO character set. See ISO 646 (ASCII).

E.4 The ordinal numbers of each value of char-type are the code values given in ISO 646 (ASCII).

E.5 The point at which the file operations REWRITE, PUT, RESET, and GET are performed, determined by the normal conventions of the operating system. Control is not returned to the program until the operation has been completed. Note that there is line by line buffering for normal interactive I/O. However, the lazy I/O ensures that prompts can be written before input is read.

E.6 The value of MAXINT is 2147483646

E.7 The accuracy of the approximations of the real operations and functions is determined by the representation (see E.2), and by the truncation of intermediate results. This gives approximately 7 decimal digits of precision.

E.8 The default value of TotalWidth for integer-type is 12

E.9 The default value of TotalWidth for real-type is 13 [illegible] The default value of TotalWidth for Boolean-type is 5

[illegible] The value of ExpDigits is 2

[illegible] The exponent character is 'E' (Upper case)

[illegible] The case used for output of the values of Boolean-type is upper case

[illegible] The procedure page outputs the form-feed character (ASCII decimal 12). The effect on any particular device depends upon that device.

[illegible] File-type program parameters should be bound to the program by the usual operating system mechanism.

[illegible] REWRITE does not overwrite previous output to the standard file output. RESET sets the file variable to the first component of the standard file output.

E.17 The equivalent symbol to '^' is implemented.
The equivalent symbol to '{' is implemented.
The equivalent symbol to '}' is implemented.

The following errors are not, in general, reported:

D.2, D.4, D.5, D.6, D.19, D.20, D.21, D.22, D.25, D.27, D.30, D.32, D.43, D.48

The following errors are detected prior to, or during execution of a program:

D.1, D.3, D.7, D.8, D.9, D.10, D.11, D.12, D.13, D.14, D.15, D.16, D.17, D.18, D.24, D.23, D.26, D.28, D.29, D.31, D.33, D.34, D.35, D.36, D.37, D.38, D.39, D.40, D.41, D.42, D.44, D.45, D.46, D.47, D.49, D.50, D.51, D.52, D.53, D.54, D.55, D.56, D.57, D.58

The processor does not contain any extensions to ISO 7185 (such extensions must be enabled by means of a compiling option, not the subject of validation).

Implementation dependent features F.1 - F.7, F.10 and F.11 of Pascal are treated as undetected errors. If the procedure page is used to write to a file then the effect of reading from that file is to read the form-feed character (F.3) The binding of variables denoted by program parameters which are not of file-type is treated as an undetected error (F.9)

Patch to Pascal Linker

This patch solves the Pascal abort error found [illegible]

1. Insert a spare cartridge into MDV1_

2. Insert a copy of Paslink into MDV2_

3. Run the following program

```
10   a = respr(25724)
20   lbytes mdv2_paslink, a
30   poke a + 3506, 67
40   poke a + 3507, 238
50   poke a + 3524, 67
60   poke a + 3525, 238
70   poke a + 3630, 67
80   poke a + 3631, 238
90   sexec mdv1_paslink,a,25724,4800
```

The new linker should now be on MDV1_ Check that this works correctly before overwriting other copies of Paslink.

[illegible] Pascal Linker

```
[illegible]
70 poke [illegible] 670,67
80 poke [illegible] 681,238
90 sexec mdv2_paslink,a/25724,1800
```

The new linker should now be on MDV1_. Check that this works correctly before over-writing other copies of Paslink.