Preface

Metacomco's QL BCPL Development Kit is a powerful package incorporating a full screen editor and a BCPL compiler. This book is intended to be a guide for users of the kit and does not aim to be fully comprehensive on all related aspects of the QL or BCPL programming. It assumes that the reader has knowledge of the QDOS operating system.

If further detailed information is required, a full specification of the Motorola 68008 microprocessor can be found in *MC68000 16/32 Bit Microprocessor Programmer's Reference Manual* (4th edition, ISBN 1-356-6795X) published by Prentice-Hall. A full description of BCPL programming can be found in *BCPL - The Language and its Compiler* by Martin Richards and Colin Whitby-Strevens published by Cambridge University Press. Further information about QDOS can be found in *QL Advanced User Guide* by Adrian Dickens (ISBN 0-947929-00-2) published by Adder Publishing.

QL BCPL Development Kit

Contents

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. See Appendix A for details of how to do this. 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 succeeds then the file is read into storage and the first few lines displayed on the screen. Otherwise a blank screen is provided, ready for the addition of new data. The message "File too big" indicates that more workspace should be specified.

When the editor is running the bottom line of the screen is used as a message area and command line. Any error messages are displayed there, and remain displayed until another editor command is given.

Editor commands fall into two categories - immediate commands and extended commands. Immediate commands are those which are executed immediately, and are specified by a single key or control key combination. Extended commands are typed in onto the command line, and are not executed until the command line is finished. A number of extended commands may be typed on a single command line, and any commands may be grouped together and groups repeated automatically. Most 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 255 characters. An empty command line is allowed, so just typing ENTER after typing 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 command line are separated from each other by a semicolon. Commands are sometimes followed by an argument, such as a number or a string. A string is a sequence of letters introduced and terminated by a delimiter, which is any character besides letters, numbers, space, semicolon or brackets. Thus valid strings might be

```
/happy/   !23 feet!   :Hello!: "1/2"
```

Most immediate commands have a corresponding extended version. See the table of commands for full details (section 1.4).

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 L /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
SB	Show block on screen
SH	Show information
SL n	Set left margin
SR n	Set right margin
ST n	Set tab distance
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: BCPL Language

2.1 Introduction

A good introduction to BCPL can be found in *"BCPL for the BBC Microcomputer"* by Chris Jobson and John Richards, published by Acornsoft, while *"BCPL - the language and its compiler"* by Martin Richards and Colin Whitby-Strevens, published by the Cambridge University Press, gives a full description of the language. The BCPL Standards Committee also provides a formal definition of the language.

BCPL provides a general structured way of expressing the logic of a program, and declaring the data required. Only a basic set of features is provided (a deliberate decision), the intention being that these basic features should be further enhanced by the definition of **procedures** as required. As a result the language is simple, compact, flexible, and efficient to implement by avoiding unnecessary overheads.

The language has been implemented on a wide range of machines ranging from large mainframes to home micros. It has been used to write compilers, assemblers, editors, database systems, applications programs, turnkey systems and games.

The first part of this chapter provides a brief description of the language as supported on the QL. The final sections describe how to run the compiler and linker. The BCPL library is described in chapter 3.

2.2 Data

Data is held in **words** which in this implementation are 32 bits long (four bytes). Unlike languages such as Pascal, a word does not have a type associated with it. The same cell can contain a pointer, an integer, packed characters, a boolean value, indeed any bit pattern. BCPL makes no checks whatsoever on the operations performed on this word, and as a result it is possible to write programs which crash the computer. On the other hand it is also possible to do anything in BCPL where in other languages one might have to resort to assembler; for example reading memory mapped I/O ports and so on.

Data can be specified in a number of ways. A decimal number is represented by a simple set of digits, if prefaced by #X then the number is in hexadecimal and if prefaced by #O or just # then the number is in octal. A character value is initialised by enclosing the character in single quotes ('); this will set the bottom eight bits only and clear the high order three bytes. The keywords TRUE and FALSE will set the entire word to the boolean value.

A **string** is a pointer to an area of memory initialised with a byte containing the length and the bytes packed in byte by byte after that; a string is represented by enclosing a set of characters in double quotes ("). A newline character is not allowed within a string, but the same effect can be obtained by the character combination *N which represents the single character newline (hex 0A). The asterisk (*) is a general escape character within strings or character values; in order to enter a real * use **. Other useful escapes are *S for space, *' and *" for ' and ", *P for form feed and *Xnn for the byte containing hex nn. Notice that 'A' is a cell containing the character A in the bottom eight bits while "A" is a pointer to two bytes, the first containing 1 and the second the character A. Note also that the expression "A"="A" will never be true because the two strings will be two different pointers to the same set of characters. The library routine COMPSTRING can be used to determine if two strings are the same.

A word can be a pointer to a sequence of other words (a **vector**) which is available for use, such an object is similar to an array. This is specified by the command VEC which is followed by a constant expression indicating the size. A vector declared by VEC n will allocate n + 1 words from 0 to n. Space is allocated for the vector within the current stack

frame, so that when the routine in which the vector is allocated returns to the calling routine the space is no longer available or valid. **Dynamic vectors** which exist for the duration of the program can be obtained from the library routine GETVEC and returned if required by FREEVEC.

A word can also be set to an initialised vector (a **table**) by means of the TABLE declaration. The keyword TABLE is followed by any number of constants separated by commas which are initialised into a vector of the required length.

2.3 Data declaration

All variables must be declared before they are used. There are four ways in which they may be declared: **manifest, local, static** and **global**. The same name may be declared several different ways, the most recent declaration which is still in scope is the one which is used. A variable name must start with a letter, and can contain letters, digits and periods (.); the name can be up to 30 characters long. Variable names must be distinct from **keywords** and upper and lower case are equated.

Manifest

Manifest values are not associated with a word at runtime, and are merely a convenient way of naming constants at compile time. They are declared within a block enclosed by block markers $(and $) and preceded by the keyword MANIFEST. The scope extends over the entire program. For example,

```
MANIFEST $( vecsize = 20 $)
```

Local

Local variables only exist within the block or procedure in which they are declared. A word on the stack is allocated for each local variable, plus any extra space required if the local is declared as a vector by VEC. The local goes out of scope when control passes out of the block in which it was declared, either by dropping or jumping out of the block or by calling another procedure. The local remains in scope if another block is entered which is enclosed within the current block; if another variable with the same name is declared the old version will be unobtainable until the new version goes out of scope. When a block or procedure is terminated the space allocated to the local is reused and the variable will be re-initialised if the block is entered again.

A local is declared by LET followed by one or more variable names, followed by an equals sign (=), followed by a matching number of expressions with which the variables are to be initialised. The list either side of the equals is separated by commas. The value ? means that the variable is not to be initialised. For example,


```
$( LET a, b = 1, (number.of.pieces * 4)
   LET q = ?
   LET v = VEC 10
   LET x, y = z, f()
   ...
```

The order in which the evaluation of expressions on the right ha side of an declaration is performed is not defined. In the example abo if z was declared global and the routine f altered z then the l declaration should be broken into two.

Static

Static data remains available throughout a program. The scope o static is the block in which it is defined, but it is possible to decla statics at the outermost level before any procedures are declared a thus ensure that the scope of a static is the entire program. Each sta declaration allocates a word from within the code space initialised to t specified constant value. A table is a vector of statics. It is go programming practice to use static values as read only. If statics a written to then the resulting code is not shareable (pure) and will not r in Read Only Memory (ROM). A procedure name which is not declared global (see below) will be a static. Statics are declared within a blo preceded by the keyword STATIC in the same format as MANIFEST.

```
STATIC $( overhead = 27 $)
```

Global

Global variables are allocated words from a special vector (the glob vector) which is available all the time that a BCPL program is runnir Each global must be allocated a unique offset in the global vector. T library routines use globals in the range 0 to 149, so user prograr should normally start allocating globals at 150. The BCPL syste header file which declares all the library globals also declares t manifest name ug which should be used to allocate globals slots u ug+1 and so on. The size of the global vector is determined when a BCI program is loaded and sufficent space allocated for that many globals. a program intends to load a code overlay whilst it is running which us higher globals it must make a dummy reference to a high global in ord to ensure that the global vector is allocated sufficient space. Globals a declared within a block preceded by the keyword GLOBAL. Each name

followed by a colon (:) and a unique global number.

```
GLOBAL $(
number.of.pieces: ug
initialise: ug + 1
$)
```


2.4 Sections

BCPL programs may be compiled as a number of sections, and the sections linked together before they are run. A section can optionally start with the SECTION directive, which is followed by a string which names the section. A section continues until the end of the file, or until a section terminator (a line containing just a period (.) at the start of the line) is encountered. In the latter case, the rest of the file up to the end of file, or another section terminator, is treated as a new section, but all the sections are automatically linked together by the compiler. If different sections are held in different files then each section must be linked together before the program is run.

The only communication between sections is by means of the global vector. Variables which are to be used in different sections must be declared global or passed as parameters. Procedures in one section which are to be called from another must also be declared global by specifying the procedure name in the global declaration list. Unless this is done procedures are defined to be static and only available within the current section.

```
SECTION "Part1"
(code of section part1)
.
SECTION "Part2"
(code of section part2)
(end of file)
```

The directive GET can be used to include part of one file within another. The most common use is to include a standard file which contains GLOBAL and MANIFEST declarations. The standard header file LIBHDR includes declarations for all the library procedures and a number of useful manifest constants. In addition a private header file can be included via GET which would normally include the declarations for globals and manifests used by different sections of a multi-section program.

Comments in BCPL code are introduced by the symbol // or ||. Anything to the right of these symbols up to the end of the line are treated as a comment. A comment which extends over many lines is enclosed by the symbols /* and */.

The layout of a BCPL program is not important as far as the compiler is concerned. Symbols should be separated by white space (spaces or tabs). Newlines can be used as white space but the following rules apply. If the compiler is expecting an expression, and the expression is complete by the time a newline is reached, then the expression is regarded as complete. This means that lines do not have to be terminated by a statement terminator semicolon (;) although multiple statements on the same line must be separated in this way. If it is wished to extend an expression over several lines then the expression must be made incomplete at the line end. This is normally achieved by either placing the entire expression in brackets, or by leaving a trailing operator at the line end. For example,

```
LET a = (number.of.pieces
         * 4 )
LET b = number.of.pieces *
        4
```

would both be correct program segments, but

```
LET a = number.of.pieces
        * 4
```

would generate a syntax error as the first line appears complete.

2.5 Procedures

BCPL programs are constructed from one or more procedures. The user must define a procedure called START which is called when BCPL execution begins. This may call other procedures until it wishes to terminate. This is done by calling the procedure STOP with a return code; this is normally zero if all went well and an operating system error code otherwise. The BCPL command FINISH is equivalent to calling STOP(0) and if START ever returns an implicit FINISH is executed.

Procedures can be declared by using LET or AND. Procedures declared using LET are only available to be called by procedures appearing textually later in the program unless they are declared global by having a matching name in the global declaration. The LET is followed by the name, an opening bracket, zero or more parameters separated by commas followed by a closing bracket. A routine is declared by following this by the keyword BE, while functions use = instead.

```
LET f(a) BE
$(
... contents of routine f, cannot call g
$)
LET g() BE
$(
... contents of routine g, which can call f
$)
```

If the first procedure is declared using LET a second procedure can be declared in exactly the same way, replacing LET with AND. In this case the two procedures may call each other. Thus in the example above, if the routine g was declared with AND instead of LET the routine f would be able to call it.

Routines declared with BE return by means of the command RETURN or by dropping out of the bottom of the routine. A function is either a simple statement, for example

```
LET f(x,y) = x+15 * y
```

or a VALOF block which will contain one or more RESULTIS commands. This is described in more detail below.

A procedure is called by specifying the name, an opening bracket, zero or more parameters and a closing bracket. Functions return a result and routines do not, although the result of a function can be ignored if this is required. Parameters are passed by value, which means that the value of a parameter is copied before it is passed to a procedure. Unlike languages such as FORTRAN, a procedure which alters its arguments does not alter the values of the parameters in the caller. Variables declared within the caller can be modified if required by passing the addresses of those variables as parameters.

2.6 Blocks

The body of a procedure consists of either a simple command such as RETURN or an assignment statement or a block. A block is enclosed by block markers $(and $) and may start with a number of local variable declarations. It will then contain a number of commands. An entire block may be used wherever a simple command is required; if the block does not contain any declarations then it sometimes known as a compound command. Thus a procedure normally consists of one or more blocks. For example,

```
LET f(x,y) BE      // Routine declaration of f
$( LET a = y*3     // Local variable a declared
   IF a=x THEN     // IF command
   $( g(a)         // Compound command
      h(y)         // .. is this block
   $)              // End of IF command block
$)                 // End of routine block
```

2.7 Expressions and Operators

An expression consists of variables, constants, VALOF blocks and functions connected by a number of operators. These operators are described in this section.

Arithmetic operators

The first set of operators refer to integer arithmetic and are as follows.

```
*      Multiply
/      Division
REM    Remainder
+      Plus
-      Minus
ABS    Absolute value
```

Logical operators

Logical operators can be used as bit by bit operators or within boolean expressions.

```
& or /\ or LOGAND   Logical AND
| or \/ or LOGOR    Logical OR
EQV                 Logical equivalence
NEQV                Not equivalent (exclusive OR)
<<                  Shift left
>>                  Shift right
NOT or ~ or~\       Logical NOT
```

If these operators are to be used as bit operators within a boolean expression then they should be bracketed. For example the first two lines below are equivalent, and test for a = true and b zero. The third line tests for a ANDed with b being zero (for example a = 3, b = 8 would yield true).

```
IF a & b = 0
IF a & (b = 0)
IF (a & b) = 0
```


Conditional operators

```
= or EQ          Equal
~= or \= or NE   Not equal
> or GT          Greater than
>= or GE         Greater than or equal
< or LT          Less than
<= or LE         Less than or equal
```

Conditional operator

There is one operator called the conditional operator which may be used to select one of two alternative expressions. The syntax is

```
<conditional expression> -> <expr1> , <expr2>
```

where if the <conditional expression> evaluates to true then value is <expr1>, and if false then the value is <expr2>. This can be used anywhere in an expression, for example

```
LET a = 15 * ( (b=2) -> c+1, d-1 )
```

which will set a to 15*(c+1) if b=2 and to 15*(d-1) otherwise. The use of brackets when using this operator is encouraged both for clarity and because of odd effects due to different operator precedence.

Addressing

BCPL provides a flexible way of handling pointers. These are used to implement features such as arrays and data structures found in other languages. A variable can be set to a pointer in a number of ways. It may be initialised as a vector with VEC or a table with TABLE or a string with "". It can also be set to point to the location at which a variable is stored. This is handled by the @ operator (LV is a synonym). If a is a variable, then @a is the address of that variable; in other words it is a pointer to where the variable a is stored

Pointers are used by the two operators ! (exclamation mark) and % (percent). If p is a pointer then !p is the value of the word which is being pointed at. If p were set to @a as above, then !p would be the value of a. If p were initialised to a table as in

```
LET p = TABLE 1, 2, 4, 8, 16
```

Then !p would be the first element of the table, i.e. the value 1.

The ! operator (often called the **indirection operator**) can also be used as a dyadic operator - taking two values. In fact !p is a shorthand form for p!0, and we can refer to p!1, p!2 and so on meaning the elements at offset 1, 2 etc. In our example above, p!1 has the value 2, p!2 is 4 and so on. We can thus use ! with a vector much as one would use a single dimension array. If p is a vector then we can access what other languages might call p(0) by p!0, and p(17) by p!17.

The indirection operator can also be used on the left hand side of an assignment statement in order to place values into the word being pointed at. For example,

```
p!17 := 25
```

will set the word at offset 17 from our pointer p to have the value 17. Beware of the lack of data types here. BCPL does not know whether a word is a valid pointer or not. If we had set p to the value 3 and then executed the above instruction we would have altered the value at (3+17); which could well crash the computer.

If you are used to languages such as BASIC then you will recognise that ! is both PEEK and POKE depending on which side of an assignment it is used. However it is not quite the same, as BCPL uses a private addressing scheme where the value stored in a BCPL pointer is the actual address divided by four. Thus if we had set p to 3 as described above, the actual value we would have altered would be (3+17)*4 = bytes 80 to 83.

There are cases where we do not wish to access a whole word from memory, merely a byte. In this case the operator % can be used. This is used in conjunction with a pointer, but takes an offset in bytes and returns only a byte. It is especially useful when the pointer was intialised to a string. If we set

```
LET p = "Hello"
```

then p%0 will be the length of the string (5), p%1 will be 'H', p%2 will be 'e' and so on. The byte operator can also be used on the left hand side of an

expression, so for example

```
p%2 := 'a'
```

would change the string stored at p to "Hallo".

It is worth noting that although P!N is the same as N!P this is not true where the % operator is concerned, i.e. P%N is not the same as N%P.

Floating point

Floating point operations are not provided in BCPL as explicit operators. This is because the representation used for a floating point value is two BCPL words, and is represented by a pointer to a vector (declared as VEC 1). Once suitable vectors have been constructed floating point operations can be performed on them by means of a set of library routines as follows.

FABS	Absolute value
FDIV	Division
FMINUS	Subtraction
FMULT	Multiplication
FNEG	Negation
FPLUS	Addition
FPOWER	Powers

These routines take real numbers in vectors and pack the result into a result vector which must also be passed as an argument. For convenience the address of the result vector is passed back as the result, enabling expressions such as

```
d := fplus( a, fminus( b, c, d ), d )
```

to be used. The values are copied before they are used, so that the same vectors can be used for each argument if required. This is useful when incrementing a real variable, for example:

```
fplus( a, inc, a)
```

Comparisons may be made on floating point numbers using the routine FCOMP which returns an integer indicating the result of a floating point

comparison. Floating point constants may be copied by copying real!0 and real!1. They may be initialised by either calling the library routine FLOAT to convert an integer into a floating point number, or by the routine FLIT which takes a string representing a floating point value and converts that into the internal representation used. Finally real numbers can be read and written from a program using the routines READFP and WRITEFP.

The floating point representation used is identical to that used in SuperBasic, and the same operating system routines are used to manipulate the values. Thus floating point calculations will have the same precision and rounding errors in either BCPL or SuperBasic.

Operator precedence

Unless otherwise altered by the use of brackets (which should always be used if you are unsure) operators bind in the following order of precedence. Operators of the same precedence evaluate from left to right.

```
!
@
%
* / REM ABS
+ -
= NE < <= > >=
<< >>
NOT
&
|
EQV NEQV
->
```


2.8 Commands

Commands are specified one to a line, or on the same line separated by semicolons (;). Commands must come after any declarations within the current block.

Assignments

An assignment statement assigns new values to one or more variables. The variables are placed in a list separated by commas, then the assignment symbol :=, then a matching number of expressions. For example

```
a := a+1
a, b := f(), z
```

The multiple assignment is exactly the same as the equivalent individual assignments.

Conditionals

The commands IF and UNLESS test a condition and execute the compound command following if the value was true or false respectively. The command TEST is followed by a condition, a compound command, the keyword ELSE and another compound command. If the condition is true the first compound command is executed, otherwise the second one is used. The condition is an expression which should evaluate to true or false, and is followed by the keyword THEN. DO is a synonym for THEN and OR is a synonym for ELSE. For example,

```
IF a=b THEN
$( a := b-1
   TEST x>16 THEN a := a-1
   ELSE $( b := 0; a := 25 $)
$)
```

For more complex cases, the SWITCHON command can be used. The syntax is

```
SWITCHON <expression> INTO
$( CASE n:
      ... Code for <expression>=n
      ENDCASE
   CASE m:
      ... Code for <expression>=m
      ENDCASE
   DEFAULT:
      ... Code for any other value
$)
```

The SWITCHON command evaluates the expression following it and matches against any values specified in the list of CASE statements (there may be one or more of these). Each CASE keyword is followed by a constant expression. If the constant matches the SWITCHON expression then control jumps to the code following the CASE. If the command ENDCASE is encountered then control jumps to the end of the block containing the CASE statements. If no ENDCASE is used then code for the next CASE will be entered. The code following DEFAULT is used if there is no match. If no DEFAULT is specified and there is no match then the effect is that of executing ENDCASE.

Flow of control

Flow of control can be altered using a large range of commands. For those whose old habits die hard, GOTO is followed by a label. A label is defined by specifying a name followed by a colon. The label will be static unless a matching global declaration exists, in which case it will be global.

Simple repetition is handled by the FOR statement. The syntax is

```
FOR <var> = <start expr> TO <stop expr> [BY <incr>]
DO <command>
```

If the BY <incr> is omitted it defaults to 1. The value of <incr> must be a constant. The <var> is automatically declared, and has the scope of the following command block. Initially <var> is set to <start expr> and the iteration test performed. If <incr> is positive then this test is satisfied if <var> is not greater than <stop expr>; if <incr> is negative then it is satisfied if <var> is not less than <stop expr>. The <command> (which may be a block) is executed repeatedly while the

test is satisfied; after each iteration the value of <incr> is added t <var>.

More generalised repetition is handled by

```
WHILE <condition> DO <command>
UNTIL <condition> DO <command>
<command> REPEATWHILE <condition>
<command> REPEATUNTIL <condition>
```

where the first forms will execute the command zero or more times, and the second forms execute it one or more times.

Any repetitive compound command may contain the commands BREAK or LOOP. LOOP jumps to just before the test of the immediately enclosing loop while BREAK jumps out of the loop. They are of particular use in conjunction with the command

```
<command> REPEAT
```

which executes <command> for ever unless BREAK, RETURN, RESULTIS or GOTO is used to exit from the loop.

VALOF blocks

A block can be made to return a result, which can be used as part of an expression. In particular, a function normally consists of a block which returns a value in this way. The block is preceded by the keyword VALOF, and it should contain one or more RESULTIS commands. RESULTIS is followed by an expression, which is the value of the block, and exits from the block. If an exit is made from a block without using RESULTIS the value is undefined. A simple example of a function might be

```
LET f(a) = VALOF
$( LET b = a-1
   IF a=0 THEN RESULTIS 0
   RESULTIS b+number.of.pieces
$)
```

VALOF is often used in conjunction with SWITCHON. In the following example the VALOF block returns a string which is used as an argument

to WRITES. The actual string used depends on the value of the variable err.message.

```
writes ("Error: ")
writes ( VALOF SWITCHON err.message INTO
               $( CASE -3: RESULTIS "No store"
                  CASE -1: RESULTIS "Not complete"
                  DEFAULT: RESULTIS "System error"
               $)

       )
```


2.9 Running the compiler

The compiler is run using EXEC or EXEC_W to load it into memory. If EXEC is used then the cursor must be attached to the new job by typing CTRL/C; in this case other commands could run at the same time (if there is sufficient memory). Using EXEC_W connects the keyboard to the new job automatically until the compiler has finished running. The compiler expects to be on drive mdv1 and it is called BCPL, so a suitable call would be

```
EXEC_W mdv1_bcpl
```

The compiler is overlaid, which means that the microdrive on which it resides must remain in place whilst the program is compiling. The compiler will normally require most of the available memory on an unexpanded QL. If a space is taken up with a SuperBasic program in memory then, although the first part of the compiler will be loaded, there may not be room for the later overlays which are larger.

The compiler asks for the source file name which must be provided. This file should contain a valid BCPL program. It next asks for a binary filename, which is where the compiled program will be placed. If just ENTER is pressed in response to this no output will be produced and the source will just be checked for syntax errors. The next question concerns the verification output. This will contain any errors detected by the compiler. If just ENTER is pressed then the screen is used, otherwise a file or a device such as ser1 can be specified. Finally any options needed are requested. The default is normally sufficient for compiling normal sized programs and this is assumed if just ENTER is pressed. The following are valid options.

Ln	Set first pass workspace to n longwords. Default 10000.
Dn	Set declaration workspace to n longwords. Default 1800.
Wn	Set code generator workspace to n longwords. Default 5000.

Enter these options sequentially on one line and then press ENTER. For example

```
L10050 D2000 W5200
```

If an invalid option is detected the question is asked again.

The next question concerns whether the window used by the compiler is to be altered. The default window will be used if the response is N or just ENTER. This is set to be the same as the initial window when the program is delivered, but it can be altered if required. See Appendix A for details of how to do this. If the response to this question is Y then the window is displayed on the screen. It can be moved by using the cursor keys and the size can be changed by using ALT and the cursor keys. When you are happy with the window position press ENTER and the compiler will start running.

Any errors detected will be displayed on the screen; if it is a syntax error then the section of program at fault will also be printed. When the compiler encounters the GET directive, a filename is constructed as follows. Firstly the string is used directly as a filename, so that if you have typed

```
GET "mdv2_myhdr"
```

then the correct file will be obtained. If a filename cannot be found, the characters "mdv1_" are placed at the start and another attempt is made. Thus a command of the form

```
GET "libhdr"
```

will read in the standard library header from mdv1. (See further in Appendix A.)

During the compilation process a work file called mdv2_bcpl_work is created and deleted. You should avoid using this filename yourself, and should ensure that there is sufficent space on the microdrive in mdv2 for both the source file and this temporary file. The temporary file is not much larger than the binary file produced and is deleted before the binary is written, so if there is room for your output there is normally room for this work file. If there are no syntax errors the output will contain a binary version of your program which you can now go on to run, after it has been linked.

2.10 Running a program

A simple BCPL program will consist of a single section, including a definition for the routine START. This program must be linked with the BCPL runtime system in order to produce a program which can be run using EXEC or EXEC_W. This is performed by the program BLINK. This is run by the command

```
EXEC_W mdv1_blink
```

and once loaded it will request the name of a binary file. This should be the output from a previous run of the compiler. 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 BCPL runtime system, and will be a code file which is directly runnable using EXEC or EXEC_W.

If you are still developing a program, the next step after creating a code file with BLINK would be to run it, and so BLINK allows a short cut here. If you simply press ENTER in response to the request for the code file, BLINK 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 BLINK and your program are in store simultaneously.

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 local declarations and vectors, and the default is 800 longwords. Therefore if your program has declared a vector of 1000 you will need to increase the stack space. In general, it is better to use the default value obtained by pressing ENTER and write your program so that large vectors are obtained from the pool of available free space by means of calls to the library routine GETVEC.

If you have written a program in several sections then you should enter the filename of the first section 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. The sections will normally be the output from the compiler, but it is also possible to include sections of program written using the assembler provided in Metacomco's Assembler Development Kit. See Appendix B for fuller details.

Chapter 3: BCPL Library

3.1 Introduction

This section describes the library routines available under QDOS, and then details the global variables which are set to useful values.

The procedures are listed in alphabetical order with a brief description. Following this a fuller description of each routine is given, again in alphabetical order.

Most routines can return a negative error code besides the expected result, and results returned should therefore always be tested before they are used. For example, the routine FINDINPUT returns either a stream control block if a file can be opened, or a negative code if the open fails. The result should always be checked before starting to read from the stream via functions such as RDCH. The negative error codes are the standard QDOS errors as described in the *QL User Guide*.

Input and Output

All I/O is performed in BCPL by viewing data as a stream of characters. A stream is created by means of a call to FINDINPUT, FINDOUTPUT or FINDTERMINAL. For more complex use, the routine OPEN is also provided.

A stream is selected for input by means of SELECTINPUT; characters are read one at a time by RDCH until the special character ENDSTREAMCH is returned which signals the end of the data. The last character read can be pushed back into the input stream by means of a call to UNRDCH although this is only guaranteed to work once without an intervening call to RDCH. An input stream may be closed when it is finished with by calling ENDREAD.

A stream is selected for output by means of SELECTOUTPUT; characters are written out to the stream by WRCH until it is closed by calling ENDWRITE. The value ENDSTREAMCH returned by RDCH should not normally be written out - it merely serves as an indicator and is not actually a character within a file.

More efficient routines are available for transferring larger amounts of data with one call. READBYTES and WRITEBYTES copy to and from a vector of characters to a stream while READFILE and WRITEFILE can be used to transfer an entire file to and from memory.

Input may be buffered or unbuffered. The routine FINDINPUT returns an unbuffered stream and FINDTERMINAL returns a buffered stream. The function OPEN can return buffered or unbuffered streams as required. The difference is important in two ways. Firstly a buffered stream reading from the keyboard will be accepted only when ENTER is pressed. This allows the user to correct the line using the normal QL controls, and also provides an automatic reflection of characters typed. This should be compared with an unbuffered stream where characters are received as soon as they are typed and are not reflected; this is required when writing an application such as a screen editor, for example.

Secondly the buffering of a stream alters the behaviour of certain streams which can be both read and written. Other BCPL implementations will normally insist that the argument to SELECTINPUT must be the result of calling FINDINPUT and the argument to SELECTOUTPUT the result of FINDOUTPUT. On the QL this is not so. A stream opened by FINDOUTPUT to the device CON can be used for input, and the stream is a valid argument to both SELECTINPUT and SELECTOUTPUT. Other streams are not valid for this two way process (it depends on the device connected to the stream) and so SELECTINPUT and SELECTOUTPUT can be used to detect if the selection is valid. If either of the routines are passed an argument for which the direction required is not valid, they will return zero and set the current stream as unset. A subsequent attempt to read or write to an unset stream will result in a fatal error, but the result from the selection routines can be tested to determine if all was well in order to avoid this.

A stream to a device such as SER1 can be used for both input and output, and BCPL allows it to be used in this way. If the routine FINDINPUT (or FINDOUTPUT) is used to open the stream then the stream will be unbuffered, which means that data can be transmitted back down the serial line as soon as a character is received. If a buffered stream were used the characters would not be made available to the calling program until a newline was received or the buffer filled up. However there may be cases where the user wishes to decide whether the stream is buffered or not, and hence the routine OPEN is provided which

allows more control over this area. The routine READLINE is also provided which always uses buffered input, this can be useful when a mixture of buffered and unbuffered is required.

Random access to file can be achieved by the routines NOTE and POINT. The first routine can be used to remember a place in a file; it returns the current byte position in the file. The routine POINT is used to position the current reading or writing 'cursor' to a given byte offset. The position might be computed as the record number times record size, for instance, or it could be the result of a previous call to NOTE. Once positioned suitably the stream may be read or written as required; streams associated with files are always read/write. Information about a file can be extracted by the routines READFILEHEADER and SETFILEHEADER.

The initial selection of streams is that input and output are both directed at the screen (the CON device). When the current output stream refers to the screen complete control of the display is available through the routines WINDOW and SCREEN. In addition the window can be recoloured using RECOLOUR and character founts changed by FOUNT. Graphics are available through the PLOT routine which takes floating point numbers as arguments. These are represented by a vector to two BCPL words (eight bytes) and are manipulated by the library routines FABS, FCOMP, FDIV, FIX, FLOAT, FLIT, FMATHS, FMINUS, FMULT, FNEG, FPLUS and FPOWER. Floating point values can be read and written using READFP and WRITEFP.

3.2 Global routines

ABORT aborts the run with diagnostic output

APTOVEC allocates a variable amount of memory from the stack

CAPITALCH converts a character to upper case

CLOSE closes a QDOS channel (see ENDREAD, ENDWRITE, OPEN)

COMPCH compares two characters irrespective of case

COMPSTRING lexically compares two strings

DATE obtains the date as a string

DELETE deletes a file

ENDREAD currently selected input stream is closed (see FINDINPUT, SELECTINPUT)

ENDTOINPUT rewinds an output stream to the beginning and reselects the stream for input

ENDWRITE currently selected output stream is closed (see FINDOUTPUT, SELECTOUTPUT)

FABS floating point absolute value (see FCOMP, FDIV, FLIT, FMATHS, FMINUS, FMULT, FNEG, FPLUS, FPOWER)

FCOMP floating point comparison (see FABS, FDIV, FLIT, FMATHS, FMINUS, FMULT, FNEG, FPLUS, FPOWER)

FDIV floating point division (see FABS, FCOMP, FLIT, FMATHS, FMINUS, FMULT, FNEG, FPLUS, FPOWER)

FINDINPUT finds and opens an input stream (see SELECTINPUT, ENDWRITE)

FINDOUTPUT finds and opens an output stream (see SELECTOUTPUT, ENDWRITE)

FINDTERMINAL opens a two way stream to a console device

FIX converts a real number to an integer (see FLOAT)

FLIT set floating point literal (see FABS, FCOMP, FDIV, FMATHS, FMINUS, FMULT, FNEG, FPLUS, FPOWER)

FLOAT converts an integer to a real number (see FIX)

FMATHS floating point maths functions (see FABS, FCOMP, FDIV, FLIT, FMINUS, FMULT, FNEG, FPLUS, FPOWER)

FMINUS floating point subtraction (see FABS, FCOMP, FDIV, FLIT, FMATHS, FMULT, FNEG, FPLUS, FPOWER)

FMULT floating point multiplication (see FABS, FCOMP, FDIV, FLIT, FMATHS, FMINUS, FNEG, FPLUS, FPOWER)

FNEG floating point negate (see FABS, FCOMP, FDIV, FLIT, FMATHS, FMINUS, FMULT, FPLUS, FPOWER)

FOUNT reset character fount

FPLUS floating point addition (see FABS, FCOMP, FDIV, FLIT, FMATHS, FMINUS, FMULT, FNEG, FPOWER)

FPOWER raise floating point to a power

FREEVEC	returns a vector allocated by GETVEC to the free pool
GBYTES	accesses bytes from memory (see GET2BYTES, GETBYTE, PBYTES)
GET2BYTES	accesses a 2 byte value from memory (see GBYTES, GETBYTE, PUT2BYTES)
GETBYTE	extracts a byte from memory (see GBYTES, GET2BYTES, PUTBYTE)
GETVEC	allocates a vector from the free store area (see FREEVEC)
INKEY	returns a character if typed within a timeout period
INPUT	determines the currently selected input stream
LEVEL	returns a value for use with LONGJUMP (see LONGJUMP)
LONGJUMP	executes a non local jump (see LEVEL)
MULDIV	evaluates (A * B)/C with a 64 bit intermediate result
NEWLINE	writes a newline to the current output stream
NOTE	returns a marker to the current position reached in a file (see POINT)
OPEN	opens a QDOS channel
OUTPUT	identifies the currently selected output stream (see FINDOUTPUT, SELECTOUTPUT)
PACKSTRING	packs the characters in a vector into a string (see UNPACKSTRING)

PLOT	screen graphics operation
POINT	resets the logical position within a file (see NOTE)
PBYTES	updates bytes in memory (see GBYTES, PUT2BYTES, PUTBYTE)
PUT2BYTES	updates a 2 bytes value in memory (see GET2BYTES, PBYTES, PUTBYTE)
PUTBYTE	updates a byte in memory (see GETBYTE, PBYTES, PUT2BYTES)
RANDOM	provides a pseudo random number
RDCH	reads the next character from the currently selected input stream (see UNRDCH)
READBYTES	reads a block of data from a file (see READFILE, WRITEBYTES)
READFILE	reads an entire file into memory (see READBYTES, SAVEFILE)
READFILEHEADER	reads a file header
READFP	reads a floating point number from the current input stream (see READN, WRITEFP)
READLINE	reads a line of data from the current input stream (see RDCH, READBYTES, READFILE)
READN	reads an integer number from the current input stream (see RDCH, UNRDCH)
RECOLOUR	recolour a window (see SCREEN, WINDOW)
REWIND	rewinds an input stream to the beginning

SAVEFILE	writes a section of memory to a file (see READFILE, WRITEBYTES)
SCREEN	screen operation (see RECOLOUR, WINDOW)
SELECTINPUT	selects a stream for input (see FINDINPUT, ENDREAD)
SELECTOUTPUT	selects a stream for output (see FINDOUTPUT, ENDWRITE)
SETFILEHEADER	writes a new file header
SETGLOBALS	initialises globals defined in loaded code
START	defines the entry point of a program
STOP	exits from a program
TIME	returns the time in seconds since the start of a job
TIMEOFDAY	returns the time of day as a string
UNPACKSTRING	unpacks the length count and characters in a string into a vector, one to a word (see PACKSTRING)
UNRDCH	pushes a character back into the currently selected input stream so that the next call of RDCH on that stream will yield that character (see RDCH)
UNSETGLOBALS	uninitialises globals defined in loaded code
WINDOW	screen operation (see RECOLOUR, SCREEN)
WRCH	writes a character to the currently selected output stream

WRITEBYTES	write block of memory to file (see READBYTES, SAVEFILE)
WRITED	writes a number to the current output stream in a given field width
WRITEF	writes out formatted data (see WRCH, WRITED, WRITEHEX, WRITEN, WRITEOCT, WRITES, WRITET, WRITEU)
WRITEFP	writes a floating point number to the current output stream
WRITEHEX	writes out a number in hexadecimal format
WRITEN	writes out a number
WRITEOCT	writes out a number in octal format
WRITES	writes a string
WRITET	writes a string in field of given width

ABORT

Purpose: To abort the run with diagnostic output.

Form: `abort( code )`

Specification:

This routine is used to terminate a program and provide diagnostic output. A message indicating the meaning of the code is printed, followed by a backtrace of the BCPL stack. This displays the functions called up to the call of ABORT and the first few values on the stack for each function. These values will be the arguments and then the local variables in the order they were defined. If a local variable is defined to be a vector then the stack will contain a pointer to the start of the vector, followed by the elements of the vector.

The routine can be called in a number of ways. Firstly it can be called by a user program, and in this case the code specified must be positive. Secondly it may be called by other library routines if a call to QDOS fails in some odd way; in this case the value of the argument will be negative and represent the QDOS error code. Thirdly the routine may be called if a runtime error such as divide by zero or calling an undefined global is detected. In this case the error code will also be negative and a suitable message will be printed.

It is possible for a user program to specify a new version of ABORT if required, which can take any suitable action. However the routine specified as ABORT should not normally return; it should either call STOP or LONGJUMP.

APTOVEC

Purpose: To allocate a variable amount of memory from the stack.

Form: `res := aptovec( fn, size )`

Specification:

The specified function is called after allocating a vector of length 'size' from the stack. The function 'fn' is called with two arguments: the first is the vector so allocated and the second is the value of 'size'. The result is any result returned by 'fn'.

This routine is provided for compatibility with other BCPL systems, but should not be used in general. In other BCPL implementations all the available space is allocated to the stack, and hence APTOVEC is used to carve up part of the stack as workspace. In this implementation most of the available space is available via the routine GETVEC.

CAPITALCH

Purpose: To convert a character to upper case.

Form: `ch2 := capitalch( ch )`

Specification:

If the character 'ch' is any of a, b,...., z then the result is the upper case version of that character (ie. A, B,...., Z respectively). Otherwise the result is just the character.

CLOSE

Purpose: To close a QDOS channel

Form: `error := close ( stream )`

Specification:

This routine is called internally by ENDREAD and ENDWRITE to close down a stream. Any streams not closed by the user program are closed automatically when the program terminates. Closing a stream makes some more store available and closes the connection with the QDOS channel. User programs should normally use ENDREAD and ENDWRITE in preference to this routine. The result is zero if the function worked and a negative QDOS error code otherwise.

See also: ENDREAD, ENDWRITE, OPEN

COMPCH

Purpose: To compare two characters irrespective of case.

Form: `res := compch( ch1, ch2 )`

Specification:

The two characters are upper cased if required (by calls of capitalch), and then lexically compared. If ch1 occurs before ch2 in the ASCII ordering then the result is negative; if they are the same then the result is zero; if ch1 occurs after ch2 then the result is positive.

COMPSTRING

Purpose: To lexically compare two strings.

Form: `res := compstring( s1, s2 )`

Specification:

The two strings are lexically compared with the characters upper cased using compch. If s1 occurs lexically before s2 then the result will be negative; if they are the same then the result will be zero; if s1 occurs after s2 then the result will be positive.

DATE

Purpose: To obtain the date as a string.

Form: `s := date()`

Specification:

This function returns a string which represents the current date (assuming that this has been set correctly when the machine was first started). The string is in the form "YYYY MMM DD"

See also: TIMEOFDAY

DELETE

Purpose: To delete a file.

Form: `error := delete ( name )`

Specification:

Attempts to delete the file specifed by the string name. If this succeeds then the result will be zero, otherwise it will be a negative QDOS error code indicating the reason for failure.

ENDREAD

Purpose: The currently selected input stream is closed.

Form: `error := endread()`

Specification:

The currently selected input stream is closed. If no stream is selected then the routine has no effect. Any store associated with the current stream is released, and the file or device connected to the stream is closed. The current input stream is unset. The result will be zero if succesful or a negative QDOS error code otherwise.

See also: ENDWRITE, FINDINPUT, FINDTERMINAL, SELECTINPUT

ENDTOINPUT

Purpose: To rewind an output stream to the beginning and reselect it for input.

Form: `error := endtoinput()`

Specification:

The currently selected output stream is reselected as the current input stream, and the current output stream is unset. The file associated with the stream is rewound to the start. This will only work on streams connected to filing system devices, and is useful when wishing to read for input a temporary file created for output during a program. The error code will be zero if the function worked, and a negative QDOS error code otherwise.

See also: REWIND

ENDWRITE

Purpose: The currently selected output stream is closed.

Form: `error := endwrite ( stream )`

Specification:

The currently selected output stream is closed. If no stream is selected then the routine has no effect. Any store associated with the current stream is released, and the file or device connected to the stream is closed. The current output stream is unset. The result will be zero if succesful or a negative QDOS error code otherwise.

See also: ENDREAD, FINDOUTPUT, FINDTERMINAL, SELECTOUTPUT

FABS

Purpose: To obtain the absolute value of a floating point number.

Form: `real2 := fabs( real1, real2 )`

Specification:

The absolute value of the real number held in the vector real1 is returned in the vector real2, whose address is returned as result. The vectors real1 and real2 may be the same.

FCOMP

Purpose: Floating point comparison

Form: `res := fcomp( real1, real2 )`

Specification:

The real numbers held in real1 and real2 are compared. The result returned is -1 if real1 is less than real2, 0 if they are equal and 1 if real1 is greater than real2.

FDIV

Purpose: Floating point division

Form: `real3 := fdiv( real1, real2, real3 )`

Specification:

The real number held in real1 is divided by the real number held in real2. The answer is returned in the vector real3, whose address is also returned as result. The vectors real1, real2 and real3 may be the same.

FINDINPUT

Purpose: To find and open an input stream.

Form: `stream := findinput( name )`

Specification:

The name passed as argument should be a string representing a valid QL file or device. An attempt is made to open the file or device for input and to construct a stream control block which may be used later by SELECTINPUT. The result is a positive value if the function worked and a negative error code from QDOS otherwise. The stream remains open until closed by a call to ENDREAD. The stream is unbuffered, so that a stream opened to CON will return characters (unreflected) as soon as they are typed. Note that in this case the cursor does not appear to be enabled. Buffered input from the console, with associated line editing, may be obtained by opening the stream with FINDTERMINAL.

See also: SELECTINPUT, ENDREAD

FINDOUTPUT

Purpose: To find and open an output stream.

Form: `stream := findoutput( name )`

Specification:

The name passed as argument should be a string representing a valid QL file or device. An attempt is made to open the file or device for output and to construct a stream control block which may be used later by SELECTOUTPUT. The result is a positive value if the function worked and a negative error code from QDOS otherwise. The stream remains open until closed by a call to ENDWRITE. Any previous file with the same name will be deleted by this call.

See also: SELECTOUTPUT, ENDWRITE

FINDTERMINAL

Purpose: To open a two-way stream to the console device.

Form: `stream := findterminal()`

Specification:

The result is this call is a stream connected to the device CON which may be used for both input and output. The stream returned may be used as the argument to both SELECTINPUT and SELECTOUTPUT. The stream associated is buffered so that data read from the terminal will not be transmitted to the program until ENTER is pressed, thus allowing line editing as supported by the QL. If unbuffered input is required a stream to CON should be opened by FINDINPUT. Note that this function does not clear the window. This means that window manipulation calls may be used to move it from its default position before anything visible happens. The result will be a positive stream or a negative error code. When a BCPL program is run the current input and output streams are both set to the result of a call to FINDTERMINAL before START is called.

See also: FINDINPUT

FIX

Purpose: To convert a real number to an integer

Form: `int := fix( real )`

Specification:

This routine converts a real number held in the vector real to an integer value, truncating if required.

See also: FLOAT

FLIT

Purpose: To convert a literal floating point to floating point.

Form: `real := flit( string, real )`

Specification:

The string should contain a character representation of a floating point number. The characters will be converted to the internal form used for floating point and this will be stored in the vector real. The vector address is returned as the result. This function is useful for initialising floating point constants. The global RESULT2 will be set to zero if the conversion succeeded.

FLOAT

Purpose: To convert an integer to a real number

Form: `real := float( int, real )`

Specification:

This routine converts an integer into a floating point number constructed in the vector real. This vector must be two BCPL words (eight bytes) long. The address of the vector is returned by the function.

See also: FIX

FMATHS

Purpose: Evaluate floating point maths operation.

Form: `real2 := fmaths( op, real1, real2 )`

Specification:

The integer operator op identifies a floating point operation to be applied to the real number contained in the vector real1. The result is placed in the vector real2 whose address is also returned as the result. The vectors real1 and real2 may be the same. Values of op are as follows.

r.acos	Arc cosine
r.acot	Arc cotangent
r.asin	Arc sine
r.atan	Arc tangent
r.cos	Cosine
r.cot	Cotangent
r.exp	Exponential
r.ln	Natural logarithm
r.log10	Logarithm
r.sin	Sine
r.sqrt	Square root
r.tan	Tangent

FMINUS

Purpose: Floating point subtraction

Form: `real3 := fminus( real1, real2, real3 )`

Specification:

The real number held in real2 is subtracted from the real number held in real1. The answer is returned in the vector real3, whose address is also returned as result. The vectors real1, real2 and real3 may be the same.

FMULT

Purpose: Floating point multiplication

Form: `real3 := fmult( real1, real2, real3 )`

Specification:

The real number held in real1 is multiplied by the real number held in real2. The answer is returned in the vector real3, whose address is also returned as result. The vectors real1, real2 and real3 may be the same.

FNEG

Purpose: To negate a floating point number.

Form: `real2 := fneg( real1, real2 )`

Specification:

The negative value of the real number held in the vector real1 is returned in the vector real2, whose address is also returned as argument. The vectors real1 and real2 may be the same.

FOUNT

Purpose: Alters the character fount.

Form: `error := fount ( fount1, fount2 )`

Specification:

Each of the two arguments is either zero or a BCPL pointer to a fount descriptor. If either argument is zero, the corresponding fount reverts to the default. Default fount 1 extends from #X20 to #X7F. Default fount 2 extends from #X80 to #XFF. A character unset in fount1 will be printed using fount2. If it is also unset in the second fount, then the lowest valid char of the second fount is used. The fount alteration applies to the screen window identified as the current output stream. Each fount is a 5x9 array of pixels in a 6x10 rectangle, and is specified by a fount descriptor which consists of a vector with the following structure. Each pixel byte contains pixels in bits 2 to 6. Bits 0, 1 and 7 MUST be unset


```
Fount%0      Initial character in fount
Fount%1      Number of characters in fount - 1
Fount%2      1st pixel byte for char 1
Fount%3      2nd pixel byte for char 1
...
Fount%10     9th pixel byte for char 1
Fount%11     1st pixel byte for char 2
...
```

FPLUS

Purpose: Floating point addition

Form: `real3 := fplus( real1, real2, real3 )`

Specification:

The real numbers held in real1 and real2 are added together. The sum is returned in the vector real3, whose address is also returned as result. The vectors real1, real2 and real3 may be the same.

FPOWER

Purpose: To raise a floating point number to a floating point power.

Form: `real3 := fpower( real1, real2, real3 )`

Specification:

The real number held in real1 is raised to the power given by the real number in real2. The result is returned in the vector real3, whose address is also returned as result. The vectors real1, real2 and real3 may be the same.

FREEVEC

Purpose: To return a vector allocated by GETVEC to the free pool.

Form: `freevec( vector )`

Specification:

A pointer to a vector allocated by GETVEC is passed as argument, and the space allocated is made available for re-use.

See also: GETVEC

GBYTES

Purpose: To access bytes from memory

Form: `val := gbytes( byteaddress, size )`

Specification:

The routine will return between one and four bytes from succesive memory locations identified as 'byteaddress'. Note that this is a byte address, not a BCPL address which is four times smaller. The value of 'size' must be in the range 1-4 and indicates the number of bytes to be returned.

Each byte is loaded separately so that the byte address need not be even, and each byte is added to those already loaded so that sign bits are propagated for values of 'size' less than 4. The routine is particularly useful when wishing to access 4 bytes not aligned to an even word boundary.

See also: GET2BYTES, GETBYTE, PBYTES

GET2BYTES

Purpose: To access a 2 byte value from memory

Form: `value := get2bytes( baseaddress, wordoffset )`

Specification:

The BCPL base address is converted to a byte address, and the value 'wordoffset' is multiplied by two and added to the address. The two byt stored there are returned as the value. The sign bit is not propagate The routine mirrors the action of GETBYTE for 2 byte values.

See also: GBYTES, GETBYTE, PUT2BYTES

GETBYTE

Purpose: To extract a byte from memory.

Form: `byte := getbyte( baseaddress, byteoffset )`

Specification:

The routine takes the BCPL base address and adds the value 'byteoffset' specified. The byte stored at this location is returned as t value. The sign bit is not propogated. This routine is provided compatibility with other BCPL implementations. The % operator BCPL provides the same function more efficiently.

See also: GBYTES, GET2BYTES, PUTBYTE

GETVEC

Purpose: To allocate a vector from the heap.

Form: `vector := getvec( size )`

Specification:

The size of the required vector is passed as argument. The result will be zero if the call failed, otherwise it will be a pointer to an area of store size+1 words in length allowing the use of vector!0 to vector!size. Space is allocated from the general heap maintained by QDOS, and the actual amount available will depend on what other tasks are running. The space will remain allocated until the end of the job, or until returned by a call of FREEVEC.

See also: FREEVEC

INKEY

Purpose: To read a character with a timeout from the currently selected input stream.

Form: `char := inkey( timeout )`

Specification:

This routine is passed a timeout in display frames (50 or 60 Hz) and attempts to read a character from the current input stream. If a character arrives within the timeout period then it is returned. If the timeout period is exceeded without a character arriving then a negative QDOS error is returned.

Note that if INKEY is being used to read characters from the keyboard, and the program is EXECed rather than EXEC_Wed, the function SCREEN(screen.cursor will be needed to enable the cursor.

INPUT

Purpose: To determine the currently selected input stream.

Form: `stream := input()`

Specification:

The input routine yields the identifier of the currently selected input stream (originally yielded by a call of FINDINPUT or FINDTERMINAL and selected by SELECTINPUT). If the current input stream is unset then the result is zero.

LEVEL

Purpose: To return a value for use with LONGJUMP.

Form: `p := level()`

Specification:

This routine returns the current value of the stack pointer, which can be used in subsequent calls of LONGJUMP.

See also: LONGJUMP

LONGJUMP

Purpose: To execute a non local jump.

Form: `longjump( p, label )`

Specification:

This routine can be used to jump out of a routine into another one further down on the stack. The value of 'p' must be the result of calling LEVEL in the destination routine. The value of 'label' will be the value of a label in the destination routine. Normally the level and label are stored in global variables and LONGJUMP used in error situations where control must return to a standard place.

See also: LEVEL

MULDIV

Purpose: To evaluate (A * B) / C with a 64 bit intermediate result.

Form: `value := muldiv( a, b, c )`

Specification:

The routine multiples 'a' by 'b' and keeps the answer accurate to 64 bits. It then divides this value by 'c' and returns the result. The remainder from the integer division is placed in the global variable RESULT2.

NEWLINE

Purpose: To write a newline to the current output stream.

Form: `newline()`

Specification:

A newline character is written to the currently selected output strean The routine is simply a call of wrch('*N').

NOTE

Purpose: To return a marker to the current position reached in a file

Form: `position := note( stream )`

Specification:

NOTE is passed a stream descriptor which must refer to a file open fo either input or output. It returns either the current relative position i the file in bytes or an error code. File positions are non-negative; erro codes are negative. The value returned would normally be use elsewhere in a call to POINT in order to reset the position in the file i order to perform random access.

See also: POINT

OPEN

Purpose: To open a stream.

Form: `stream := open( name, mode, bufflen )`

Specification:

This opens a stream for the file or device specified as name in the requested mode. The result of the function is either a stream control block or a QDOS error code. If bufflen is zero the SCB is set up for unbuffered (character by character) reading. If bufflen is greater than zero the SCB is set up for line buffered reading with the specified buffer length.

The mode represents the way in the file should be opened as follows.

```
0   exclusive use of an old file
1   shared use of an old file
2   exclusive use of a new file
3   exclusive use of an overwritten file
4   open as a directory
```

This function is used internally by FINDINPUT, FINDOUTPUT and FINDTERMINAL which should normally be used in preference.

See also: CLOSE, FINDINPUT, FINDOUTPUT, FINDTERMINAL

OUTPUT

Purpose: To identify the currently selected output stream.

Form: `stream := output()`

Specification:

This routine yields the identifier of the currently selected output stream (originally yielded by a call of FINDOUTPUT or FINDTERMINAL and selected by SELECTOUTPUT). If the current output stream is unset then the result will be zero.

See also: FINDOUTPUT, FINDTERMINAL, SELECTOUTPUT

PACKSTRING

Purpose: To pack the characters in a vector into a string.

Form: `size := packstring( vector, string )`

Specification:

The zero element of the vector should contain the number of characters to be packed; this is ANDed with a mask of 255 (#XFF) to prevent the string being longer than 255 characters.

The specified numbers of characters are then packed into the string such that

```
vector ! i -> string % i
```

etc. The length is inserted, and any unused bytes in the last word of the string are set to zero. The string and vector may be coincident (ie., @vector = @string), but they may not otherwise overlap.

The result is the number of words used in the string.

See also: UNPACKSTRING

PLOT

Purpose: Plot lines and arcs on the screen.

Form: `error := plot( code, a1, a2, a3, a4, a5 )`

Specification:

This is a generalised graphics routine which takes a code value and up to five floating point arguments, each of which is passed as a pointer to a vector of two BCPL words (eight bytes). The possibilities for code are one of the following integers; suitable manifest names are defined in the standard header file LIBHDR.

`plot( plot.point, x, y )`
Plot a point at position x,y.

`plot( plot.line, xs, ys, xf, xf )`
Plot a line starting at xs,ys and finishing at xf,yf.

`plot( plot.arc, 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.

`plot( plot.ellipse, x, y, e, r, angle )`
Plot an ellipse centered at x,y with eccentricity e and radius r. The value of angle indicates the rotation angle.

`plot( plot.scale, ly, x, y )`
Set the origin as x,y with length of vertical axis ly.

`plot( plot.cursor, yo, xo, y, x )`
(Note that the syntax is odd, but the order is correct). Position the cursor at point (x+xo, y-yo). The values of xo and yo are in pixels and allow the cursor to be offset from the current graphics point.

POINT

Purpose: To reset the logical position within a file.

Form: `error := point( stream, position )`

Specification:

POINT takes a stream descriptor which must refer to a file but which may be open for input or output. The current position in the file is reset to the byte offset specified by the second argument. This can either be computed or the result of a previous call to NOTE. The result is zero if all went well and a negative error code otherwise. If the error code returned indicates end of file then the file will have been repositioned as far as possible in the relevant direction. Thus a call with a very large number as the second argument will move to the end of the file.

See also: NOTE

PBYTES

Purpose: To update bytes in memory.

Form: `pbytes( byteaddress, size, value )`

Specification:

The 'size' bytes stored at the specified byte address are replaced by the requested number of bytes from the 4 byte 'value'. This is the opposite of GBYTES.

See also: GBYTES, PUT2BYTES, PUTBYTE

PUT2BYTES

Purpose: To update a 2 byte value in memory.

Form: `put2bytes( baseaddress, wordoffset, value )`

Specification:

The least significant 2 bytes from 'value' are stored at the address given by the combination of the BCPL 'baseaddress' and the 2 byte offset 'wordoffset'. This is the opposite of GET2BYTES.

See also: GET2BYTES, PBYTES, PUTBYTE

PUTBYTE

Purpose: To update a byte in memory.

Form: `putbyte( baseaddress, byteoffset, value )`

Specification:

The least significant byte of 'value' is stored at the memory address given by the BCPL 'baseaddress' and the byte offset 'byteoffset'. This is the opposite of GETBYTE, and is the same as the use of the % operator in BCPL.

See also: GETBYTE, PBYTES, PUT2BYTES

RANDOM

Purpose: To provide a pseudo random number.

Form: `randominteger := random ( seed )`

Specification:

The 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.

RDCH

Purpose: To read the next character from the currently selected input stream.

Form: `ch := rdch()`

Specification:

A call of RDCH yields the next sequential character of the currently selected input stream, unless that stream is exhausted, in which case the end-of-stream character (ENDSTREAMCH) is returned. The action of RDCH may be altered by intervening calls to UNRDCH.

Any exceptional condition encountered by the runtime system while handling any reading function will result in a fatal error and a call to ABORT. This covers attempting to read from an invalid stream and so on.

See also: UNRDCH

READBYTES

Purpose: To read characters from the current input stream.

Form: `m := readbytes( v, n )`

Specification:

READBYTES reads characters from the input stream into the buffer v. At most n characters are read. The number actually read is returned as m. If m < n then there are no more characters in the file. If READBYTES is called after end-of-file has been reached it returns zero.

See also: READFILE, READLINE, WRITEBYTES

READFILE

Purpose: To read an entire file into memory.

Form: `result := readfile( stream, buffer, length )`

Specification:

This reads an entire file contents into the buffer specified, which should be of the given length. The stream descriptor should refer to a filing system device opened for input. A call to READFILEHEADER may be used to obtain the length of a file before it is read into memory. The result is either a negative error code, or the number of bytes actually transferred (which should be equal to length).

See also: READFILEHEADER, SAVEFILE

READFILEHEADER

Purpose: To read the header associated with a file.

Form: `result := readfileheader(stream, buffer, length)`

Specification:

READFILEHEADER reads the file header into the buffer indicated. The result is either a negative error code or the number of bytes read. The value of length is the length of the buffer in bytes which should be at least 14. This will only work on streams connected to files on filing system devices. The values returned in the buffer are as follows.

```
Buffer!0      Length of file in bytes
Buffer%4      Access byte (normally zero)
Buffer%5      File type
Buffer%6      1st byte of info
Buffer%13     8th byte of info
Buffer%14     Zero
Buffer%15     Length of file name
Buffer%16     1st character of file name
```

The file type is 0 for data files, 1 for EXECable binary files and 2 for Linker format binary files. If the file type is 1 then the first four bytes of the info field contains the size of the data area to be allocated when the program is EXECed.

See also: READFILE, SETFILEHEADER

READFP

Purpose: To read a floating point number from the current input stream.

Form: `bool := readfp( real )`

Specification:

The routine reads a floating point number from the current input stream in the same fashion as READN. The format may optionally include a decimal point and use of E notation. The real number is packed into the two word vector indicated by real, and returns TRUE if a valid floating point number was encountered and FALSE otherwise. The routine reads one extra character which it returns to the input stream via a call of UNRDCH.

See also: READN, UNRDCH, WRITEFP

READLINE

Purpose: To read a line into a buffer.

Form: `m := readline( v, n )`

Specification:

READLINE attempts to read a line (terminated by ASCII linefeed) into the buffer v. At most n characters are read. If a linefeed is read, the result, m, will be positive. It is a count of the number of characters read (including the terminating linefeed). If a linefeed has not been read, m will not be positive. -m characters have been read, but the line is not complete. This can be because the buffer was not large enough, or because the file is exhausted. Note that the last character in a file need not be a linefeed. Calls of READLINE after end-of-file has been reached return zero.

READLINE always uses buffered input, so that a line typed at the terminal in response to this routine will be reflected and can be altered before ENTER is pressed, even if the stream was created by FINDINPUT and is hence unbuffered.

See also: READBYTES, RDCH

READN

Purpose: To read a number from the current input stream.

Form: `n := readn()`

Specification:

Characters are read from the currently selected input stream using RDCH until one is found that is not a space, a newline, or a tab character. Unless the first character found is a digit or a plus or a minus sign, the result is zero, the global RESULT2 is set to -1 and the character is returned to the input stream via a call to UNRDCH.

A number is then read comprising the following contiguous string of digits. There is no check for numeric overflow. The number is terminated by the first non-digit encountered, and this character unread. The result is the number, and the global RESULT2 is set to zero.

Note that a plus or minus sign which is not followed by a digit will be read as zero.

This definition of READN differs from some others in that the last character read is pushed back into the input stream via a call of UNRDCH.

See also: RDCH, UNRDCH

RECOLOUR

Purpose: Recolour a window.

Form: `error := recolour( colours )`

Specification:

The window identified by the current output stream has each of the colours replaced by an alternative. The single argument is a pointer to an 8-byte area. Each byte contains a colour number in the range 0 - 7 representing the new colour required.

See also: SCREEN, WINDOW

REWIND

Purpose: To rewind an input stream.

Form: `error := rewind()`

Specification:

The current input stream is rewound so that the next character read is taken from the start of the file. Note that REWIND only works on channels to filing system devices. A QDOS error code is returned on failure, and zero if all went well.

See also: ENDTOINPUT

SAVEFILE

Purpose: To save a section of memory as an entire file.

Form: `result := savefile( stream, buffer, length )`

Specification:

The specified number of characters from the buffer are written to the output stream which should refer to a filing system device. The result is either a negative error code, or the number of bytes actually transferred (which should be equal to length).

See also: READFILE, SETFILEHEADER, WRCH, WRITEBYTES

SCREEN

Purpose: Screen handling

Form: `error := screen( code, arg1, arg2 )`

Specification:

This is a generalised operation for handling the QL screen. It always refers to the window specified as the current output stream. The type of operation is determined by the code. Many operations require no further arguments, some require one, a few require two. The error return will be zero if all went well and a negative QDOS error code otherwise.

The possibilities for code are one of the following integers; suitable manifest names are defined in the standard header file LIBHDR.

`screen( screen.border, colour, width )`
Set window border to the specified colour and width. The border is inside the window limits and is doubled on the vertical edges.

`screen( screen.cursor )`
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.

`screen( screen.nocursor )`
Disable the cursor.

`screen( screen.at, column, row )`
Position the cursor at the specified row and column, using character coordinates.

`screen( screen.atp, 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.

`screen( screen.tab, column )`
Tab to column specified.

`screen( screen.newline )`
`screen( screen.left )`
`screen( screen.right )`
`screen( screen.up )`
`screen( screen.down )`
Move the cursor to the start of the next line, or one space in the relevant direction.

`screen( screen.scroll, dist )`
`screen( screen.scroll.top, dist )`
`screen( screen.scroll.bottom, 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.

`screen( screen.pan, dist )`
`screen( screen.pan.line, dist )`
`screen( screen.pan.eol, 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.

`screen( screen.clear )`
`screen( screen.clear.top )`
`screen( screen.clear.bottom )`
`screen( screen.clear.line )`
`screen( screen.clear.eol )`

Clear the screen, or part of it, to the current paper colour. Part screens are defined as in scroll and pan above.

`screen( screen.paper, colour )`
`screen( screen.strip, colour )`
`screen( screen.ink, colour )`

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

```
screen( screen.flash, switch )
screen( screen.underline, switch )
screen( screen.fill, 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.

```
screen( screen.mode, 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.

```
screen( screen.size, 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.

See also: RECOLOUR, WINDOW

SELECTINPUT

Purpose: To select a stream for input.

Form: `stream := selectinput( stream )`

Specification:

The stream, which should be the result of a call of FINDINPUT or FINDTERMINAL, is selected and made the current input stream. If the stream is valid then the stream will be returned, if not zero will be returned and the current input stream will be unset.

See also: FINDINPUT, FINDTERMINAL, ENDREAD

SELECTOUTPUT

Purpose: A stream is selected for output.

Form: `stream := selectoutput( stream )`

Specification:

The specified stream, which should be the result of a call of FINDOUTPUT or FINDTERMINAL, is selected and made the current output stream. If the stream is valid then the stream will be returned; if not, zero will be returned and the current output stream will be unset.

See also: FINDOUTPUT, FINDTERMINAL, ENDWRITE

SETFILEHEADER

Purpose: To set the header associated with a file.

Form: `result := setfileheader( stream, header )`

Specification:

The stream must refer to a filing system device which has been opened for output. The header is a vector containing 14 bytes of information to be placed in the file header. The result is either a negative QDOS error code, or 14 or 15. 14 indicates that 14 bytes were set in the header; 15 indicates that the stream was a serial stream rather than a file, and a $FF followed by the 14 bytes have been sent. The header vector should contain a file header as described under READFILEHEADER; only the first 14 bytes can be set by this call.

SETGLOBALS

Purpose: To set globals defined in a code buffer.

Form: `bool := setglobals( buffer, length )`

Specification:

This assumes that the region indicated by buffer contains a concatenation of BCPL modules. The length is the length of the buffer in bytes. The buffer would normlly be filled by a call of READBYTES or READFILE. The routine scans the code and sets up the correct global values for any global routines defined within the loaded code segment thus providing access to the routines so loaded. This mechanism allows a flexible overlay system to be constructed. Once the code is no longer required the globals defined within the code segment can be set to the unset value by a call to UNSETGLOBALS.

The result of SETGLOBALS is TRUE if the global vector was big enough and FALSE if not; in order to load a code segment which uses globals greater than those used in the root program a dummy global large enough must be used in order to force a larger global vector allocation.

See also: READBYTES, READFILE, UNSETGLOBALS

START

Purpose: To define the entry point of a program.

Form: `start()`

Specification:

The routine START is defined by the user and indicates the main routine of a BCPL program. Once the BCPL runtime system has initialised itself it will call START; if this ever returns the routine STOP is called.

See also: STOP

STOP

Purpose: To exit from a program.

Form: `stop( returncode )`

Specification:

This causes the program to stop, passing the return code back. QDOS will then clear up any open streams and GETVECed space. If the program was run by EXEC_W rather than EXEC and returncode is a negative QDOS error code then a text message associated with the error code is displayed. A zero returncode indicates no error condition. The BCPL command FINISH, or returning from START causes STOP(0) to be called.

See also: START

TIME

Purpose: To return the time in seconds since the start of a job.

Form: `res :=time()`

Specification:

When a BCPL 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 inital time.

TIMEOFDAY

Purpose: To obtain the time of day as a string.

Form: `s := timeofday()`

Specification:

This function returns a string which represents the current time (assuming that this has been set correctly when the machine was first started). The string is in the form "HH:MM:SS".

See also: DATE

UNPACKSTRING

Purpose: To unpack the length count and characters in a string into a vector, one to a word.

Form: `unpackstring( string, vector )`

Specification:

The length and characters in the string are unpacked one to a word into the vector, such that


```
string % 0 -> vector ! 0
```

etc. The string and vector may be coincident (ie., @string = @vector) but they may not otherwise overlap.

See also: PACKSTRING

UNRDCH

Purpose: To push a character back into the currently selected input stream so that the next call of rdch on that stream will yield that character.

Form: `bool := unrdch()`

Specification:

UNRDCH attempts to put the last character read back into the stream so that it may be read again. It may not succeed. The result indicates whether or not it has.

UNRDCH will always succeed if the last call for the stream was a read. UNRDCH will always fail for a stream which has just been opened or rewound or one to which POINT has been applied. Repeated calls of UNRDCH will only work on streams with input buffers. It is never possible to step back beyond the beginning of the current line. If the input buffer is not as large as the current line, it may not be possible to get back this far.

See also: RDCH

UNSETGLOBALS

Purpose: To return globals to their unset pattern.

Form: `unsetglobals( buffer, length )`

Specification:

The buffer with specified byte length must contain a concatenation of BCPL object modules. This routine sets the global values of routines defined in the buffer to an unset value which can be detected by the runtime system; an unset global will cause ABORT to be called. An overlay may be loaded via READBYTES or READFILE and the globals initialised by a call to SETGLOBALS; when the overlay is finished with the globals should be set back by a call to UNSETGLOBALS.

See also: READBYTES, READFILE, SETGLOBALS

WINDOW

Purpose: Screen operations on windows.

Form: `error := window(code, descriptor, colour, width)`

Specification:

This is a general purpose routine for manipulating windows. The first argument, code, describes the action to be taken. The second argument is a vector of four BCPL words and is used to specify the window. The first two words are a width and a height; the last two are an x coordinate and a y coordinate, x being measured to the right and y down from some origin. The last 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. Width is only used when defining a new window.

`window( window.askp, descriptor )`
`window( window.askc, descriptor )`
Return the size of the window in the first two words and the cursor position relative to the top left corner in the last two words. window.askp returns the information in pixel coordinates; window.askc returns it in character coordinates.

`window( window.define, descriptor, colour, width )`
Define a new window as specified by the descriptor. The size is given in pixels by the first two words in the descriptor. The position, also in pixels, in the last two words 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.

`window( window.fillblock, descriptor, colour )`
Fill a block in a window. The size of the block is given in pixels by the first two words in the descriptor. The top left corner of the block is given in pixels by the last two words. The third argument, colour, is only relevant for this call of window. The block is filled with the specified colour (or stipple) according to the current overprinting mode.

See also: RECOLOUR, SCREEN

WRCH

Purpose: To write a character to the currently selected output stream.

Form: `wrch( ch )`

Specification:

The character is written to the currently selected output stream. As with the reading functions, all exceptional conditions encountered during output (such as drive full) result in fatal errors.

See also: SAVEFILE, WRITEBYTES

WRITEBYTES

Purpose: To write characters to the current output stream.

Form: `writebytes( v, n )`

Specification:

This routine writes n characters from the vector v to the current output stream. An exceptional condition will result in a fatal error. The output is not terminated by a newline character, although one can of course be placed in the buffer or explicitly written by a call to NEWLINE.

See also: NEWLINE, SAVEFILE, WRCH

WRITED

Purpose: To write a number to the current output stream in a given field width.

Form: `writed( n, w )`

Specification:

The integer n is written in a field of w spaces, except that if the field is not wide enough then it will be expanded to the necessary size. The number will be right justified in the field, and the field will be padded with blanks.

WRITEF

Purpose: To write out formatted data.

Form: `writef( format, a, b, ..., k )`

Specification:

The 'format' is a string. This is written out character by character as f WRITES except that when a percent (%) character is encountered, th next item from 'a', 'b', etc., is taken, and written out in a forma dependant on the character directly following the %. In some cases th format character is followed by a length character which should be 0,... or A,. .,F representing the numbers 0 to 15.

The options are as follows :

%S	Write argument as a string.
%Tn	Write argument as a string in a field of n places.
%C	Write argument as a character.
%Fn	Write argument as floating point in a field of n places.
%On	Write argument in octal in a field of n places.
%Xn	Write as for %O but in hexadecimal.
%In	Write as for %O but in decimal.
%N	Write in decimal.
%$	Ignore this argument.

For the cases of writing in a given field width, the truncation an padding schemes are as for the corresponding direct routines (ie., write writeoct, etc.).

If the character following the % is not one of the above then tha character is written out. Hence to output a % the string 'format' shoul contain %%.

Note that strings that are written out (%S and %Tn) are not processed for percent symbols.

See also: WRCH, WRITED, WRITEFP, WRITEHEX, WRITEN, WRITEOCT, WRITES, WRITET

WRITEFP

Purpose: To write a floating point number to the current output stream.

Form: `writefp( real, w )`

Specification:

The floating point number in the vector real is written in a field of w spaces, except that if the field is not wide enough then it will be expanded to the necessary size. The number will be right justified in the field, and the field will be padded with blanks.

See also: READFP

WRITEHEX

Purpose: To write out a number in hexadecimal format.

Form: `writehex( n, w )`

Specification:

The number 'n' is written to the currently selected output stream in hexadecimal format in a field of 'w' spaces. If the field is not wide enough then only the lowest order 'w' digits are written. Otherwise the number is right justified in the field and padded with zeroes.

See also: WRITEOCT

WRITEN

Purpose: To write out a number.

Form: `writen( n )`

Specification:

The number 'n' is written to the currently selected output stream in a field that is just wide enough. The routine is simply a call of WRITED(n, 0).

See also: WRITED

WRITEOCT

Purpose: To write out a number in octal format.

Form: `writeoct( n, w )`

Specification:

The specification of this routine is exactly that of WRITEHEX, except that octal format is used.

See also: WRITEHEX

WRITES

Purpose: To write a string.

Form: `writes( string )`

Specification:

The string is written to the currently selected output stream, character by character, using the routine WRCH.

WRITET

Purpose: To write a string in a field of given width.

Form: `writet( string, w )`

Specification:

The string is written to the currently selected output stream in a field of 'w' spaces. If the field is insufficiently wide then it is expanded as necessary, otherwise the string is left-justified in the field and the field is padded with blanks.

3.3 Global Variables

CIS	The current input-stream.
CODEBASE	A pointer to the start of the main code segm
COS	The current output stream.
GLOBSIZE	The size of the global vector. This is glo zero, so the address of globsize (@globsi refers to the base of the global vector.
RESULT2	This value holds extra information af certain function calls.
STACKBASE	A pointer to the start of the stack.
SYSIN	The default input stream.
SYSOUT	The default output stream.

Appendix A: Installation

Changing the default window

Both the editor and the BCPL compiler allow the window which is to be used 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.

Changing the default drive name

The BCPL compiler uses five overlays which it assumes are all located on drive mdv1 along with the file LIBHDR. For those users who upgrade their QLs with disc drives, there is the possibility of changing the default drive to something other than mdv1. This means that the compiler, its overlays and LIBHDR can be copied from the supplied microdrive to disc, so that the compiler can be EXECed from the extra device. This option will not be given when installing the editor ED since it does not use overlays and hence can be EXECed from any device.

The INSTALL program

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

```
LRUN mdv1_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. This is similar to the mechanism used when altering the window during normal program initialisation. 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 the BCPL compiler 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.

In the case of the BCPL compiler the program will then go on to ask for a default drive name where it should look for its overlays and LIBHDR. If you do not wish to change the default drive name the reply should be

```
MDV1
```

(Note - the reply must not be MDV1_). If you do wish to change the default drive name the reply should be the device name, for example

```
FLP1
```

In this latter case the compiler will append 'FLP1_' to the name of each of 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 of the editor or the compiler. It is unlikely to be useful with any other program except these two.

Appendix B:

Assembler linking conventions

A BCPL program consists of a number of sections linked together with the runtime system before they are run. Normally these sections are written in BCPL, but it is possible to write one or more of them in assembler. The assembler code must have a certain format as described below.

Firstly all assembler routines must be position independent; the normal output of the Metacomco QL Assembler is suitable. Secondly the assembler code module must have the same structure as that produced by the BCPL compiler. This structure consists of the length of the module in long words, followed by the code, followed by global initialisation information aligned to a long word.

All communication in BCPL is performed through the global vector, and hence an assembler routine must define at least one global by which it may be called. Globals are defined by entering pairs of values at the end of the code, preceded by a value of zero to indicate the end of the list, and followed by a number representing the highest global referenced in the program. Each pair consists of the value of the global number, and the offset of the label from the base of the program segment. The structure would be as follows.

```
Length in words
Assembler code
...
...
0
Global number of routine n
Offset of routine n
Global number of routine n-1
Offset of routine n-1
...
...
Global number of routine 1
Offset of routine 1
Highest referenced global
```


An assembler routine must use registers in the same way as BCPL, at least initially. If registers are altered from the standard described here they must be saved and restored before returning or calling another BCPL routine. A BCPL routine is passed up to four arguments in the registers D1 to D4. (The first argument being in D1, the second in D2 etc.). These arguments can also be found on the BCPL stack at 0(A1) to 12(A1) as can any further arguments which are placed on the BCPL stack at 16(A1) onwards. If a routine returns a result it is returned in the register D1. The register D0 is used as the stack increment (in bytes) of the calling procedure. This value will be 4 bytes for each local variable in the calling routine plus a further 12 bytes to allow for information stored on the BCPL stack at call time. Otherwise all data registers are available as work registers.

Address registers are used as follows. A0 is always set to zero, to allow the use of data registers as pointers by using the construction (A0,Dn.L). If A0 is corrupted it should be restored by

```
SUBA.L Z,Z
```

A1 is, as mentioned above, the BCPL stack pointer, while A2 points to the global vector. Both of these are machine addresses. (A1) is the first stack location, 4(A1) the second and so on. The value of global variable 20 is held in 80(A2). These must be saved and restored if they are corrupted. A3 is used as a link register during function call and return, and is otherwise available as a work register. A4 is used as a base register, and is set to point to the base of the current routine. It is then possible to use this register to address locations in the current program segment. A4 is loaded with the function address during function application. A4 can also be used as a general work register. A5 and A6 are used to point to the code which performs function application and function return respectively. They should both be restored if they are altered. Function call is executed by

```
JSR (A5)
```

with the registers set up as detailed above. Return is accomplished by

```
JMP (A6)
```

BCPL uses long word addresses, and thus machine addresses must be converted before calling BCPL routines. Any address passed to BCPL

must therefore be aligned to a long word. The assembler directive

```
CNOP 0,4
```

must be used to ensure that this is the case. In particular, entry points defined as global and the global initialisation information at the end of a module must be long word aligned.

Example program

The following is a trivial example program which defines global 150 (FRED), and which calls global 73 (WRITES) to write out a string. Finally the routine calls a QDOS service to find out the current display mode, and returns this as result. Note the use of registers and the use of CNOP to ensure alignment.

```
*
* Example assembler routine
*
       RORG    $0
FRED   EQU     150              Define global 150
WRITES EQU     73               Use this to write
*                               a string
*
FIRST  DC.L    (ENDMOD-FIRST)/4 Length of module
*
       CNOP    0,4              Ensure long word
*                               aligned
LAB    LEA.L   MESS,A3          Point to string
       MOVE.L  A3,D1            Place into D1
       LSR.L   #2,D1            Convert to BCPL
*                               address
       MOVEQ   #12,D0           Minimum stack frame
       MOVEA.L (WRITES*4)(A2),A4 Extract address
*                               from global
       JSR     (A5)             Call WRITES, arg
*                               in D1
*
       MOVEQ   #$10,D0          QDOS mode request
       MOVEQ   #-1,D1           Read mode
       MOVEQ   #-1,D2           Read display type
       MOVEM.L A0-A2,-(SP)      Save BCPL registers
```



```
       TRAP    #1               QDOS call
       MOVEM.L (SP)+,A0-A2      Restore registers
       JMP     (A6)             Return, result in D1
*
* Constant string
*
       CNOP    0,4              Ensure long word
*                               aligned
MESS   DC.B    6,'Hello',$0A    "Hello*N"
*
*Global information
*
       CNOP    0,4              Ensure long word
*                               aligned
       DC.L    0                End of global list
       DC.L    FRED,(LAB-FIRST) Global number
*                               and offset
       DC.L    WRITES           Highest global
*                               referenced
ENDMOD END
```

Appendix C: Example programs

Example Program 1

An example of how to read a string in BCPL.

```
SECTION "Example 1"

GET "LIBHDR"

MANIFEST $( length = 10 $)

LET START ( ) BE
$( LET v = VEC length
   LET no.of.chars = ((length + 1) *
                     // since !0 to !length are
                     // available
                        BYTESPERWORD) - // now in bytes
                           1            // less one byte
                     // which will hold the length count
   $( LET res = readstring(v,no.of.chars)
                                  // read a string
      WRITEF( ((res=-1) -> "*NString too long",
                           "*NString = %S*N"),v )
                                  // write it out
   $) REPEATUNTIL COMPSTRING(v,"stop")=0
                                  // continue until
                                  // 'stop' typed
$)

// Routine to read up to 'n' chars and place them in
// a vector 'v'. Initial tabs, spaces CR's are ignored.
// A tab, space or CR is treated as a terminator.

AND readstring(v,n) = VALOF
$( LET ch    = ?
   LET count = 0              // initialise counter

   WRITES("*NEnter a string : ") // Prompt for input
```



```
   ch := RDCH()                  // get first char
   WHILE breakchar(ch) DO ch:= RDCH()
                                 // eat leading spaces,
                                 // tabs and CR's

   UNTIL breakchar(ch) DO        // space, tab or CR
                                 // terminates string
   $( count := count + 1         // increment counter
      v%count := ch              // put char in vector
      ch := RDCH()               // get next character
      IF count=n THEN BREAK      // exit loop if
                                 // array full
   $)

   v%0 := count                  // store string length

   IF (count=n) THEN             // If array is full
   $( IF ~breakchar(ch)          // and the next char is
                                 // not '*S','*T'
         THEN count := -1        // or '*N', return -1 to
                                 // indicate string too long
      UNTIL breakchar(ch) DO ch := RDCH()
                                 // Throw away any excess
                                 // chars
   $)

   UNRDCH()                      // Put the 'break'
                                 // char back

   RESULTIS count
$)

// Routine which returns TRUE if the argument
// past to it is a space, tab or CR, and
// FALSE otherwise

AND breakchar( c ) = (c = '*S')|(c = '*T')|(c = '*N')
```

Example Program 2

This example shows the use of floating point and writef.

```
GET "LIBHDR"

MANIFEST $( err.bad.number  = 1
            err.dosum       = 2
         $)

LET error(code,arg1,arg2) BE
$( SWITCHON code INTO
   $( CASE err.bad.number  :
      WRITEF("%S not a number. Error = %N*N",arg1,arg2)
      ENDCASE
      CASE err.dosum :
      WRITEF("Bad op %C in dosum*N",arg1)
      ENDCASE
      DEFAULT          :
      WRITEF("Unknown Error = %N*N",code)
   $)
$)

// myflit is the same as flit but checks for errors
// during conversion

LET myflit(s,v) BE
$( FLIT(s,v)
   IF RESULT2 < 0 THEN error(err.bad.number,s,RESULT2)
$)

// dosum works out which operation +,-,* we want to do,
// does it and works out result

LET dosum(op,arg1,arg2,res) BE
$( LET fun = ?       // function to be called
   SWITCHON op INTO
   $( CASE '+'  : fun := FPLUS ; ENDCASE // plus
      CASE '-'  : fun := FMINUS; ENDCASE // minus
      CASE '**' : fun := FMULT ; ENDCASE
                  // times (see note on **)
      DEFAULT   : error(err.dosum,op)
```



```
                  // bad operator give error
               RETURN                  // and return
   $)

// use WRITEF to write out args and result

   WRITEF("%F8 %C %F8 = %F8*N",arg1,op,
            arg2,fun(arg1,arg2,res))
$)

LET START() BE
$( // Declare some space for floating point numbers
   LET n1 = VEC 1
   LET n2 = VEC 1
   LET n3 = VEC 1

   SCREEN(screen.clear) // clear the screen

   // Try some test conversions

   myflit("1234.567",n1)
   myflit("4321.756",n2)
   myflit("abcdef",n3) // this should be an error

   // Try some operations

   dosum('+',n1,n2,n3) // add numbers
   dosum('-',n1,n2,n3) // subtract numbers

   // NOTE BCPL treats * as a special character
   //  (e.g. *N, *S) so we have to use ** for *

   dosum('**',n1,n2,n3)// times numbers
   dosum('~',n1,n2,n3) // this should be an error
   error(27)           // bad error call
$)
```

Example Program 3

This example program displays the time. Use EXEC to run, CTRL-C to get the window cursor keys to move it where you want, and <RETURN> to start displaying the clock.

```
SECTION "clock"                 // section name

GET "LIBHDR"                    // get library procedures


// manifests to make program easier to read

MANIFEST $( chars.in.clock = 8
            height.of.char = 10
            border.width   = 1
            width          = 0
            height         = 1
            xcoord         = 2
            ycoord         = 3
            down           = #XD0
            up             = #XD8
            left           = #XC0
            right          = #XC8
            //cursor control keys
         $)

GLOBAL $( chsize : ug $)        // reserve space
                                // in global vector
```



```
LET START() BE
$( LET descriptor = VEC 3  // get 4 bcpl words on stack
   initialise(descriptor)  // set up window
   movewindow(descriptor)  // move window around
   SCREEN(screen.nocursor) // disable cursor
   showtime()              // display time
$)

/*
```

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). It then defines a window that is big enough to display the clock and a white border.

```
*/
AND initialise(descriptor) BE
$( WINDOW(window.askc,descriptor)
   TEST descriptor!width > 40 THEN chsize := 6
                              ELSE chsize := 12
   descriptor!width  := chsize * (chars.in.clock+2) +
                        (border.width*4)
   descriptor!height := height.of.char +
                        (border.width << 2)

// the next two line define the initial position of the
// clock

   descriptor!xcoord := 50
   descriptor!ycoord := 10
$)

// this routine allows the user to move the window
// about using the cusor keys until the enter key
// is pressed

AND movewindow(descriptor) BE
$( LET ch = ?
   LET w,h = descriptor!width,descriptor!height
   LET x,y = descriptor!xcoord,descriptor!ycoord
   LET stream = SELECTINPUT(FINDINPUT("CON"))
   SELECTOUTPUT(stream)
```

```
   SCREEN(screen.cursor)
   $( descriptor!xcoord := x
      descriptor!ycoord := y
      SCREEN(screen.clear)
                // clear screen
      SCREEN(screen.border,black,border.width)
                // remove old border

// redraw window with a white border

      WINDOW(window.define,descriptor,white,
             border.width)
      ch := RDCH()
      SWITCHON ch INTO
      $( CASE  '*N': RETURN   // carriage return
         //down
         CASE  down: IF y < 1  LOOP
                     y := y-1
                     ENDCASE
         //up
         CASE    up: IF h+y > 256 LOOP
                     y := y+1
                     ENDCASE
         //left
         CASE  left: IF x < 1  LOOP
                     x := x-1
                     ENDCASE
         //right
         CASE right: IF w+x > 512 LOOP
                     x := x+1
                     ENDCASE
         DEFAULT: LOOP
      $)
   $) REPEAT
$)

AND showtime() BE
$( SCREEN(screen.border,white,1)  // draw white border
   SCREEN(screen.at,1,0)          // home cursor
   WRITES(TIMEOFDAY())            // write out time
$) REPEAT
```