QC USER MANUAL

CONTENTS

1 INTRODUCTION

QC is a version of C specially produced for the Sinclair QL. It is a subset of the standard C language as described in Kernighan & Ritchie. It contains all the features of RatC (Appendix 1 of Berry & Meekings) plus the following extras:

switch, for, do, goto statements
logical operators **&& ||**
unary operators **! ~**
comma expressions
assignment operators
short / long integers
unsigned values
initialised local variables
static variables & functions

All the language features are described in more detail in section 3.

You are recommended to read this manual thoroughly before using the compiler.

1.1 Purpose and Scope of this Manual

The QC User Manual provides an introduction to the QC compiler on the Sinclair QL. It includes all the information you need to write programs using the compiler, but does not attempt to teach the C language.

1.2 Conventions Used in this Manual

When QC keywords, operators or variables are mentioned in the text, they will be in **bold**. UPPERCASE will be used for SuperBASIC commands and QDOS filenames.

When describing the language in section 3, syntax examples will have keywords and other explicit language elements in bold, but generic elements will be in angle brackets, eg:

do { <statement> } **while** (<expression>);

The library routines described in section 4 all have the start of the function declaration, giving the function names, its parameters and the types of all of the parameters. This is all printed in bold.

The use of the word "module" in this manual refers to a group of subroutines compiled together in one go, no matter how many source files go into it; It can refer to the assembler source file or the relocatable relocatable binary file or the C source file(s).

1.3 QC Components List

The QC compiler comes with the following components:

* two microdrive cartridges containing the compiler, assembler, linker, runtime library files and example programs
* two blank microdrive cartridges for making backup copies
* a copy of "A Book on C" by R. E. Berry & B. A. E. Meekings
* an A5 ring binder containing this manual

The microdrive tape labelled QC1 contains the following files:

QC	the QC compiler
QCASM	the QC assembler (used as pass 2 of the compiler)
LINK	the QL Linker
QC_LINK	the linker control file
BACKUP	a program to copy this tape

The microdrive tape labelled QC2 contains the following files:

QC_LIB	the standard library file containing all routines defined in section 4 of this manual
QDOS_LIB	the extra QDOS library file containing all routines defined in section 5 of this manual
STDIO.H	the standard I/O header file (see section 4)
COMPILE	a C program to drive the compiler, assembler and linker (See section 2.4)
WINDOW_MGR	a C program for adjusting programs' default windows
COMPILE_C	the source of the COMPILE program
BACKUP	a program to copy this tape

In order to have enough space for your own programs, you should copy the library files and the header file onto one microdrive, which will then have room for your files. Copy the others onto a different microdrive which will be used less frequently.

1.4 What You Will Need to Use the Compiler and Write C Programs

This compiler will run on any QL, and does not need any add-on RAM. You will need a text editor program to create source files, for example the ED program as provided with the Sinclair Assembler.

If you want to write large systems, floppy disks will be useful because of the extra storage space, but microdrives can be used for programs of up to a few hundred lines.

If you do not have a text editor, you can use QUILL to create your program sources. The files produced by QUILL cannot be read directly by the compiler: you must first print the text into a print file, using an appropriate printer driver. (Try pagelength 0, linelength 80, LF line separator with no preamble or postamble.)

1.5 Making Backup Copies

It is strongly recommended that you make backup copies of the supplied tapes before using them. The program BACKUP is supplied on each tape and can be used to copy the tape. We suggest you then use the new tapes for running the compiler and save the originals as backups.

To run the BACKUP program, put the tape to be copied in MDV1 and a blank tape in MDV2 then enter the command:

```
EXEC_W MDV1_BACKUP
```

The program will ask for the source and destination directory names if you just press ENTER it will assume source is MDV1_ and destination MDV2_. It will ask whether you want to format the blank tape. If you do not format the tape, it will ask whether you want to overwrite files on the backup tape if they already exist. This option is useful when you make backups of your working tapes. It will then copy the tape.

1.6 Other Useful Manuals and Books

The C programming Language
B. W. Kernighan and D. M. Ritchie (Prentice-Hall)

A C Reference Manual
S. P. Harbison and G. L. Steele (Prentice-Hall)

The C Programming Tutor
L. A. Wortman and T. O. Sidebottom (Prentice-Hall)

Note that some of the example programs in these books will not work with QC because they use features of C not supported by the compiler.

1.7 Disclaimer

Under no circumstances will GST Computer Systems Limited be liable for any direct, indirect, incidental or consequential damage or loss including but not limited to loss of use, stored data, profit or contracts which may arise from any error, defect or failure of the QC compiler software.

GST Computer Systems Limited has a policy of constant development and improvement of their products. We reserve the right to change manuals and software at any time and without notice.

1.8 Copyright and Trade Marks

2 HOW TO RUN THE COMPILER

The QC compiler takes C program sources (created with a text editor) and converts them to assembler language text. This must then be assembled using the QC assembler (the Sinclair Macro Assembler can also be used). This produces a relocatable binary file. One or more relocatable binary files are then combined with the runtime library by the linker, to create an executable program file.

2.1 Compiler

The compiler program is called QC. Put the tape marked QC1 (or a copy of it) in MDV1 and your data tape (which is created from a copy of QC2) in MDV2, then type the command:

```
EXEC_W MDV1_QC
```

The compiler will start up and ask for a command line. You can now type the filename of the C program you want to compile - this is the simplest command line.

If the filename you give is MDV2_MYPROG the compiler will first look for MDV2_MYPROG_C and then for MDV2_MYPROG if the _C file cannot be found.

The compiler will produce an output file named after the source program: using the example above, it will create MDV2_MYPROG_ASM

You can put more filenames on the command line, in which case all of them are read as input, and the name of the first file is used to create the output filename.

You can also include a number of options on the command line. These all start with a dash followed by a letter (upper or lower case):

-M monitor: write the first line of each function to the screen as it is compiled.

-A alarm: the compiler will bleep whenever it prints an error message to the screen.

-P pause: after printing an error message to the screen, the compiler will wait for you to press the ENTER key before continuing.

-C comments: the C code is included in the output file as comments, and the assembler code is formatted neatly.

-D <dir> directory: the specified directory is searched for include files. Any device or directory name can be specified here: eg -D MDV2_

-L <name> listing: compiler listing output is sent to the named file or device.

If a blank line is given to the compiler as a command line, it will reprompt for another command line. If no filenames are given, but the command line does contain some options, it will take data typed in at the keyboard as the compiler source, and send its output to the screen.

If there are any compilation errors, the compiler will return a "not complete" error code to QDOS.

2.2 Assembler

The output from the compiler is in assembler source format. An assembler is included with the compiler. Run the assembler by typing:

```
EXEC_W MDV1_QCASM
```

When the assembler asks for a command line, type the name of your file (the final _ASM can be omitted) eg:

```
MDV2_MYPROG
```

This will then generate a binary file named after the source file - in this case, MDV2_MYPROG_REL.

To produce a listing at the assembler stage, include in the command line the option -LIST <filename>. See appendix C for a list of the assembler options.

If any errors are reported by the assembler it will return a "not complete" status to QDOS. Check for functions or variables with the same name as 68000 registers. You can ignore any warnings from the assembler.

2.3 Linker

The linker is then used to combine your program with the runtime library. A linker control file is included on QC1, called QC_LINK. Run the linker by typing:

```
EXEC_W MDV1_LINK
```

and when the linker asks for a command line, type the name of the binary file (the final _REL can be omitted) followed by the name of the control file, and any other linker options, eg:

```
MDV2_MYPROG MDV1_QC_LINK -NOLIST
```

This will produce a program MDV2_MYPROG_BIN which can then be executed using EXEC or EXEC_W. See appendix C for a list of linker options.

Note that the linker control file assumes that the library file QC_LIB is on the tape in MDV2_. You will have to edit the control file to change this arrangement.

If there are any errors or warnings from the linker, or any multiply defined or undefined symbols, do not try to run the program.

2.4 The Control Program COMPILE

A program called COMPILE is included on the tape to drive the compiler, assembler and linker together. It will ask for the name of the file to be compiled, and will construct command lines for each program. You should leave out the _C and just enter the name of the program:

```
EXEC_W MDV2_COMPILE
```

Press F1 to select the compile option:

```
Name of file to compile?      MDV2_MYPROG
```

Note that the compile program can also be used to execute other programs including the Window Manager and Backup programs.

The source of this program is included with the compiler as an example program, and to allow you to change it if you wish. If you do recompile this program, you must then adjust its data requirement down to 2k in order to leave enough memory for the compiler. (You can do this using the window manager program - see 2.6.)

2.5 Linking More Complicated Programs

To link more complicated programs involving more than one module, you should create your own link control file. If you want to specify all the modules named in the control file, take the supplied control file MDV1_QC_LINK and replace the line "INPUT *" with an input directive for each of the C modules to be linked together. Then run the linker with a command line like this:

```
-WITH MDV2_MYCONTROL_FILE
```

Alternatively, if you still want to name one of the modules in the link command line, leave in the INPUT * line which tells the linker when to read the module named in the command line. Follow this line by an INPUT directive for all the other modules.

The linker control file should look like this:

```
SECTION   S.HEADER
SECTION   S.GBASE
SECTION   S.GLOB
SECTION   S.CCODE
SECTION   S.RBASE
SECTION   S.RELOC
SECTION   S.TRAILER
INPUT     MDV2_MYPROG1_REL
INPUT     MDV2_MYSUBS_REL
INPUT     MDV2_OTHERCODE_REL
LIBRARY   MDV2_QDOS_LIB
LIBRARY   MDV2_QC_LIB
DEFINE    G$ = C.GLOBBA
DEFINE    M$ = C.ENDGLO + $8000
DATA      8K
```

If you have a complicated program which uses lots of stack, you may want to change the DATA directive in the control file to change the workspace used by the program. If you do not use any of the routines listed in section 5 of this manual, you can delete the line which calls for library file QDOS_LIB to speed up the link operation. Other parts of the control file should not be changed.

You can create a subroutine library by appending a number of relocatable binary files together. If this is included in a link with an INPUT directive, all the modules will be included in the link. If a LIBRARY command is used, the file will be searched, and only those modules which resolve an undefined external reference will be extracted.

2.6 The Window Manager Program

This program is included to allow you to adjust the default window used by any program written in C or any of the utilities included on the tape.

```
EXEC_W  MDV2_WINDOW_MGR
```

The program will ask for the name of the program to adjust. The window position and size can be adjusted interactively, and the program asks for paper and ink colour. The windows can be defined with tags: the top line of the window is reserved to display the program name.

This program also allows you to adjust the stack space reserved by a program. You should not change this for any of the supplied programs.

When you run a program with a tagged window, it will be displayed during program initialisation. If the window is not tagged, it is not cleared by the C program until it is written to. This allows filter programs to be created which do not create a screen window.

2.7 Using the Compiler with Floppy Disks

If you have floppy disks on your QL, you can easily transfer the programs across to take full advantage of the increased storage space and speed.

None of the programs will need modification to run on floppy instead of microdrive - the "compile" program actually checks to see if FLP1 exists and if not it defaults to MDV1.

You will need to change the linker control file which names the library files. This is an ordinary text file which can be updated with a text editor.

3 The QC Language

This section defines the language facilities provided by the QC compiler. It is not intended to teach the C language: you are recommended to read the book included with the compiler or one of the other books listed in the introduction if you want to learn how to program in C.

C comments start with **/*** and end with ***/** and they cannot be nested.

3.1 Variables and Types

Variables are the fundamental data objects manipulated in a program. All variables in QC must be declared before they are used.

3.1.1 Variable Names

Variable names are made up from letters, digits, and the underline character, and they must start with a letter. Only the first eight characters in a variable name are significant.

Uppercase and lowercase are treated as different by the compiler: it is conventional in C to use lowercase for variable names, and uppercase for symbolic constants (see 3.6.2).

Some names are reserved, including all of the keywords (eg **if**, **else**, **int**) and names beginning with "**c_**" are reserved for use within the runtime library. The names of the 68000 registers (eg A0, D0, USP) are also reserved.

3.1.2 Declarations

Variables can be declared globally (outside of any function) in which case they are in scope from the declaration to the end of the module, or locally in which case the variable is private to the block or routine.

The types supported by QC are:

int	32 bits wide
long	32 bits wide (same as int)
short	16 bits wide
char	8 bits wide
pointers	32 bits wide

A declaration specifies an optional storage class specifier followed by an optional type specifier followed by a list of variable names.

Global variables can have the following storage classes:

(none) the variable is available outside the module in **extern** directives.

extern the variable must be declared (without a storage class specifier) in some other module. (See 3.1.4)

static the variable is private to the module.

The storage class specifier is ignored on local variables. The allowed classes are **auto** and **register**. It is not normally specified. The names are accepted for compatibility with other compilers. Storage classes **static** and **extern** are not accepted by QC for local variables.

The types **short** and **long** can be used on their own, or they can be followed by **int**.

An asterisk before the name creates a pointer to objects of that type, whereas square brackets are used to define arrays and enclose the array size.

Examples:

```
int lower, upper, step;
long value;
short int word;
char c, buffer[512], *str;
```

Note that when defining an array of **N** locations they are accessed using offsets 0 to **N-1**.

3.1.3 Initialising Variables

Local variables can be initialised with any expression (but local arrays cannot be initialised).

Global **int** and **char** variables can initialised to constant values, for example:

```
int  i = 0,
     j = (345*678)+123;
char c = '?';
```

Initialised and uninitialised variables can be mixed in the same declaration. Global arrays can be initialised by giving the initial values in curly brackets:

```
short int array[10] = { 0, 1, 2, 3 };
```

The number of initialised array elements can be less than the size of the array, in which case the rest of the array is uninitialised. The array size can be left out, in which case the number of initialised elements is taken as the array size.

A character array or character pointer can be initialised with a string constant, for example

```
char
     *pointer = "a string",
     string[] = "another string";
```

In C, strings are arrays of characters terminated by a zero byte.

3.1.4 External Variables

Global variables declared in other modules can be referenced using the **extern** directive. This may not be used inside functions, as it is treated like a global variable definition. The symbol is then in scope until the end of the module.

If an array is declared external you do not specify the array size, but use an empty pair of square brackets.

External variable directives cannot have initial values: the variable can be initialised in the module which declares it.

example:

```
extern int
     qwerty, *pointer,
     datarray[];
```

3.1.5 Unsigned Values

The keyword **unsigned** can be put at the start of any declaration, to indicate that the variables are to be treated as unsigned numbers. Unsigned arithmetic is then used in expressions including unsigned variables. Examples:

```
short i;
unsigned short u;

i = 40000;          /* overflow: treated as -25536 */
u = 40000;

i = i/2;            /* result is -12768 */
u = u/2;            /* result is 20000 */
```

char	values range from	-128	to	127
short	values range from	-32768	to	32767
int	values range from	-2147483648	to	2147483647
unsigned char	values range from	0	to	255
unsigned short	values range from	0	to	65535
unsigned int	values range from	0	to	4294967295

3.2 Operators and Expressions

3.2.1 Constants

The constants accepted by QC are:

decimal numbers eg	-1, 76890, 1000000000, -999
octal numbers eg (starting with a zero digit)	012, 0377
hex numbers eg (start with 0x)	0x0a, 0xFF
character constants eg	'a' 'DE'
string constants eg	"I am a string" ""

Note that character constants actually generate **int** values, and can have one or two characters in them, whereas a string constant is actually an array of characters terminated by a null byte: the value of a string constant is the address of the array.

String and character constants interpret the backslash character specially to specify some characters:

\n	is a newline code
\f	ascii FORMFEED
\t	ascii TAB
\b	ascii BACKSPACE
\'	a single quote (for use in character constants)
\"	a double quote (for use in strings)
****	the backslash character itself
\123	(up to 3 digits) interpreted in OCTAL
\0	(special case of above) null
\x12	(up to 2 digits) interpreted in HEX

3.2.2 Expressions

Some operators require an "lvalue" operand. An lvalue is an expression referring to an object: a variable name is the simplest case, but array indexing and pointer indirection are also lvalues.

When (signed) **char** or **short** values are used in expressions, they are sign-extended to **int**. When **unsigned char** or **short** values are used in expressions they are padded with zeroes to **unsigned long**.

A summary of operator precedence is given at the end of this section (3.2.8).

Overflow is always ignored when evaluating an expression.

3.2.3 Primary Expressions

Constants, variables and strings are all primary expressions.

Any expression in parentheses is a primary expression. Parentheses are used in this way to control expression evaluation order.

A primary expression followed by an expression in square brackets is a primary expression. This is used to index arrays.

A primary expression followed by a sequence of zero or more expressions enclosed in parentheses is a function call, all of which is a primary expression.

3.2.4 Unary Operators

Unary operators group right-to-left. The unary operators are:

`*` indirection: the expression is normally a pointer, and the result of the expression is the value pointed to.

`&` address-of: this can only be applied to an <lvalue>, and returns the address of the lvalue.

`-` unary minus: gives the 2's complement of the expression

`!` logical not: returns 0 (FALSE) if expression is nonzero (TRUE), or 1 (TRUE) if the expression is zero (FALSE).

`~` ones complement: bitwise complement

`++ --` increment and decrement: these can be prefix or postfix operators. The operand must be an <lvalue>.

If ++ is used in prefix mode, the value is incremented and the result of the expression is the new value. If ++ is used in postfix mode, the value is incremented and the result of the expression is the original value.

— works similarly.

3.2.5 Binary Operators

All binary operators group left-to-rignt, so if an expression contains operators of equal precedence, they are parsed from the left. For example:

```
a - b + c       is parsed as        (( a - b ) + c )
```

The operators in this section are listed in order of decreasing precedence, but operators of equal precedence are grouped together.

If either operand to an operator is **unsigned**, then both operands are regarded as unsigned, unsigned arithmetic is used and the result is unsigned.

```
*    multiply
/    divide
%    remainder
```

Multiply overflow is ignored. Division truncates towards zero: the sign of the remainder is the same as the sign of the divisor. It is always true that:

```
((a/b)*b + a % b) == a          /* Provided b is nonzero */
```

```
+    add
-    subtract
```

If an integer value is added to (or subtracted from) a pointer, it will be scaled by the base type of the pointer, so if **p** is a pointer into an array, **p+1** is a pointer to the next object in the array.

If two pointers to objects of the same type are subtracted, the result is scaled to give the number of objects between the two pointers. This is only useful if toth pointers point into the same array.

```
<<   shift left
>>   shift right
```

Shift left fills with zero bits. Shift right (unsigned) fills with zero bits, but preserves the sign bit on signed values.

<	less than
<=	less than or equal
>=	greater than or equal
>	greater than
==	equal
!=	not equal

These operators return 1 (TRUE) if the relation is true and 0 (FALSE) if the relation is false. Unsigned comparisons are made if either operand is unsigned.

&	bitwise and
^	bitwise exclusive or
|	bitwise inclusive or

These operators perform a bitwise operation on the values and return an integer result.

&&	logical and
||	logical or

The logical operators test whether values are zero or nonzero, and return a value which is zero or one. They guarantee left-to-right evaluation of operands, but the right operand is not evaluated if the left operand determines the result.

3.2.6 Assignment Operators

Assignment operators group right-to-left, and the result of the operation is the value assigned. It is therefore possible to perform multiple assignments, so for example

```
*fred = bert[1] = thing = 0;
```

sets **thing** to zero, then sets a location in array **bert** to zero then sets the object pointed to by **fred** to zero.

```
<lvalue> = <expression>
```

This is simple assignment: the expression is evaluated and the result saved in the variable or location specified by the lvalue.

When assigning to a **char** the value is truncated to 8 bits and higher bits are lost. Values are truncated to 16 bits for assignment to **short** objects.

Other assignment operators perform some arithmetic on an object:

<lvalue> <op>= <expression>

<op> can be any one of:

+ - * / % >> << & ^ |

Note there can be no space between the <op> and the equals symbol.

The behaviour of a <op>= b
is equivalent to a = a <op> b
but a is only evaluated once.

3.2.7 Comma Expressions

<expression> , <expression>

A pair of expressions separated by a comma are evaluated left-to-right. The value of the first expression is discarded, and the expression on the right gives the value of the whole expression.

Comma expressions are typically used where the syntax requires an expression, but a number of side effects are required, for example in the control expression of a while loop:

```
while ( ++ptrl, fred += increment, c = *ptrl, c != EOF )
    print ( c );
```

Note that when the context imposes another meaning on the comma character (eg in the list of parameters to a function) a comma expression should be enclosed in brackets to avoid confusing the compiler, for example:

```
func ( a, ( b=2, b+3 ), c );
```

3.2.8 Operator Precedence Summary

(starting with most binding):

```
<primary> ( <expression>s...)   function call
<primary> [ <expression> ]      array indexing

* <expression>                  indirection
& <lvalue>                      address of
- <expression>                  unary minus
! <expression>                  unary logical not
~ <expression>                  unary bitwise complement
++ <lvalue>                     pre increment
-- <lvalue>                     pre decrement
<lvalue> ++                     post increment
<lvalue> --                     post decrement

* / %                           multiplicative operators

+ -                             additive operators

<< >>                           arithmetic shifts

< <= => >                       inequalities
==  !=                          equality operators

&   ^   |                       bitwise logical operators

<expression> && <expression>    logical and
<expression> || <expression>    logical or

= += -= *= /= %= >>= <<= &= ^= |=   assignment operators
<expression> , <expression>     comma operator
```

3.3 Control flow and Statements

A note on semicolons: in C, semicolons are not used to separate or terminate all statements: they are part of the syntax of certain statements.

3.3.1 Expression Statement

<expression> ;

An expression statement is an expression followed by a semicolon. Usually expression statements are assignments or function calls. Examples:

```
a = 1;
ptr++;
func ( a, b, c );
```

3.3.2 Compound Statement (Block)

{ <declarations> <statements> }

A compound statement allows several statements to be used where the language syntax requires one statement. It also allows local variables to be defined.

A compound statement is a pair of braces (curly brackets) containing zero or more declarations followed by zero or more statements. For example:

```
{    int a, b;

     a = getval();
     b = getval();
     putval ( a + b );
}
```

3.3.3 Conditional Statement

The two forms of the conditional statement are:

if (<expression>) <statement1>

if (<expression>) <statement1> **else** <statement2>

In both cases, the expression is evaluated. If it is true (nonzero), statement1 is executed. In the second case, if the expression is false (zero) statement2 is executed.

As there is no "endif" construction in C, an **else** is always associated with the most recent incomplete **if** statement.

3.3.4 While Statement

while (<expression>) <statement>

The expression is evaluated and if true (nonzero) the statement is executed and the loop repeated until the expression becomes false.

3.3.5 Do Statement

do <statement> **while** (<expression>);

The statement is executed then the expression is evaluated. If the expression is true (nonzero) the loop is repeated.

3.3.6 For Statement

The **for** statement has the form:

```
for ( <expression1> ; <expression2> ; <expression3> )
     <statement>
```

This is functionally equivalent to:

```
<expression1>;
while ( <expression2> )
{
     <statement>
     <expression3> ;
}
```

The first expression is the loop initialisation; the second is the loop termination test, evaluated before each iteration; the third expression is the loop re-initialisation, often incrementing some counter.

Any one of the three expressions may be left out: if expression2 is left out, the loop is executed continuously - see the **break** statement in 3.3.8. For example:

```
for ( count=0 ; ; ++count )
{    int c;
     c = getchar();
     if ( c == EOF ) break;
     putchar(c);
}
```

Variables used in a **for** statement must be predeclared, unlike some other languages.

3.3.7 Switch Statement

The switch statement has the form:

switch (<expression>) <compound-statement>

The expression is evaluated, and the result is used to select which part of the compound statement is executed.

Each statement in the compound statement can be labelled with one or more case prefixes:

case <constant-expression> :

There can also be one default prefix, of the form:

default:

The value of the switch expression is compared with each case constant. If one of the case constants matches, control is passed to the statement after that prefix. If no case constant matches, and there is a default prefix, control is passed to the statement after the default prefix. If no constant matches and there is no default then none of the statements in the block are executed.

See the **break** statement (3.3.8) for exiting a **switch** statement. If the break statement is not used, execution of the code will "drop through" past labels into the following statements.

Declarations are not allowed in the compound statement of a switch statement.

Example:

```
switch ( ch )
{

case 'a': case 'e': case 'i': case 'o': case 'u':
     print ( "Its a vowel" );
     break;

case 'y':
     print ( "Sort of a vowel" );
     break;

case 'f': case 'h': case 'l': case 'r': case 's':
     print ( "Soft consonant" );
     break;

default:
     if ( isalpha ( ch ) )
          print ( "Consonant" );
     else print ( "Not a letter" );
     break;
}
```

3.3.8 Break, Continue and Return

The statement

break;

exits the current loop or switch statement.

The statement

continue;

causes the rest of the current loop to be skipped, and starts another iteration of the loop.

The statements

return;
return <expression> ;

exit the current function. In the first case the result of the function is undefined. In the second case the expression is evaluated and returned as the function result.

3.3.9 Goto Statement

The goto statement has the form:

goto <label> ;

where the label is defined in the current function like this:

<label> :

A label name is constructed in the same way as a variable name.

QC does not allow **gotos** into or out of blocks which have local variables.

3.3.10 Null Statement

The null statement consists of just a semicolon. It is often used in loops where the whole work of the loop is done by side-effects of the control expression, as in this example:

```
while ( *dest++ = *source++ )
     ;
```

Here the bytes pointed to by **source** are copied to **dest** and both pointers are incremented. The loop is terminated when a null byte is transferred, as the result of the assignment is then zero.

3.4 Functions and Program Structure

3.4.1 Program Structure

A C program is built up from a number of functions. (There is no distinction in C between functions and procedures - all functions return a value, but the value can be undefined, and the value returned by a function call can be ignored.) There is no "main program" in C, but each program has one function called **main** which is called when the program is executed.

A C program source file therefore contains a number of function declarations, together with optional global or external variable declarations (see 3.1.2) and/or optional preprocessor directives (see 3.6).

3.4.2 Function Declarations

```
<name> ( <arg-list>... ) <arg-declarations>...
{ <declarations>... <statements>... }
```

A function declaration is constructed as above. It starts with the function name (function names are constructed in the same way as variable names) followed by parentheses enclosing the list of arguments.

arg-list is a list of zero or more argument names (separated by commas). These are the names by which the arguments are known within the function. The names are in scope for the whole function.

arg-declarations is a series of declarations specifying the types of the arguments. Each argument must be declared as either int or char or a pointer to one of these two. If the name is prefixed by an asterisk it names a pointer to the type specified, eg:

```
afunction ( i, j, ch, str )
     int i, *j;
     char ch, *str;
     { .... }
```

In this example, i is declared to be an integer, j is a pointer to an integer, ch is a char and str is a pointer to char.

An array is passed to a function as the address of the first element in the array. As an alternative to declaring an array argument type as a pointer (using an asterisk) it can be declared using a pair of square brackets, eg:

```
func ( array ) int array[];
```

In this case, the parameter **array** is still a pointer to the base of the array, but the syntax suggests to the programmer that an array is to be processed.

After the argument declarations comes the function body, which must be a compound statement.

A function which has no parameters is declared as follows:

```
func () { ... }
```

3.4.3 Function Calls

To call a function, the function name is followed by a list of parameters, in parentheses. Each parameter can be any expression:

```
afunction ( value()-123, &intarray[17], '0'+digit, string );
```

Each parameter expression is evaluated, and a copy of the value is made before the function is called. It is this copy which is used by the function as its parameter. Arguments are therefore passed by value. If you want a function to be able to update some variable, you pass the address of the variable to the function:

```
main()
{
     int a, b;
     :
     swap ( &a, &b )      /* call function to swap variables */
     :                    /* note addresses are passed */
}


swap ( addra, addrb )     /* this is the swap function */
     int *addra, *addrb;
     /* the parameters are defined as pointers */
{
     int temp;

     temp = *addra;
     *addra = *addrb;     /* the locations pointed to */
     *addrb = temp;       /* get modified here */
}
```

If a function has no parameters, it is called by following the function name with an empty pair of parentheses, eg:

```
func();
```

3.4.4 Function Results

All functions return results, but the result returned by a function is undefined unless the function executed a **return** statement with a return value (3.3.8).

In QC, all functions return an **int** value. Pointers can be returned from functions because a pointer variable is the same number of bits as an integer variable, and the value is unchanged by QC when a pointer is assigned to an integer or vice-versa.

3.4.5 Static Functions

If a function is declared to be **static** its name is not available to other modules, just like a static variable. The function name is just preceded by the word **static** in the declaration:

```
static <name> ( <arg-list>... ) <arg-declarations>...
{ .... }
```

3.4.6 External Functions

If a function exists in another module, it can be declared as external as follows:

```
extern int <name> ();
extern int ( * <name>  ) ();
```

(Note that the function list is not declared here.) The word **int** can be left out of the declaration, as the default type in an **extern** declaration is **int**. The second form of the declaration is treated as equivalent to the first by QC, but full C compilers treat it as declaring a pointer to a function.

At the end of a module, any undefined symbols are assumed to be external functions.

3.4.7 Parameters which are Function Addresses

It is possible to call a function and pass as one of the parameters (the address of) another function which it in turn calls. This can be done in QC using the following syntax in the declaration and in the call:

```
( * <name> )
```

For example, **funcl** is a function which takes a function parameter and is declared like this:

```
funcl ( arg )  int (* arg) ();
     /* arg is a pointer to a function */
{
     (* arg)();      /* the function is called like this */
}
```

The call to funcl with parameter func2 looks like this (the function is named on its own with no parentheses):

```
funcl ( func2 );
```

3.4.8 Functions with a Variable Number of Parameters

Normally, the number of parameters passed to a function when it is called should be the same as the number of parameters declared for the function.

Some functions (eg **printf** in the library - 4.5.1) require a variable number of arguments.

If you want a function with a variable number of parameters (like **printf**) it can find the number of parameters passed by calling the library routine **ccargc()**. (This routine must be called at the start of the function before any other functions are called.) Using this and the information that the arguments are stored as **int**s in reverse order on the stack, it is possible to access all the arguments correctly. Note this is not portable. Example:

```
func ( arg )          /* declare one arg for the function */
     int arg;         /* as an integer */
{
     int  numargs,    /* this will hold the number of args */
          *argp;      /* this will be used as a pointer to them */

     numargs = ccargc();              /* get the number of args */

     argp = &arg + numargs-1;         /* point at the first arg */

     while ( numargs-- )              /* for each arg, in order, */
          process ( *argp-- );        /* process it & step to next */
}
```

See section 6 for more information on the arrangement of function parameters on the stack.

3.5 Pointers and Arrays

3.5.1 Pointers

Variables which hold the addresses of other variables are called pointers. They provide a very powerful and flexible mechanism for processing data.

Pointers are typed according to the type of object being pointed to. This is because **int**s take four bytes of memory but **char**s take only one byte. Pointers are manipulated according to the size of object pointed to: if you add one to a character pointer, it then points to the next character, and if you add one to an integer pointer it points to the next integer, although this is four bytes wide.

Any value added to or subtracted from an integer pointer is therefore scaled by the compiler to allow for the size of integers.

When the difference is taken between two pointers, the result is the number of objects between them, not the number of bytes. (If the pointers are not of the same type, or do not point into the same array, the results are undefined.)

The syntax for declaring a pointer is the same as the syntax for declaring integer and char variables, except that the variable name is prefixed with an asterisk:

```
<type>    * <name>;
```

If a pointer is used in an expression on its own, the address is used or altered according to the expression. If the pointer is preceded by an asterisk, the value pointed to will be used in the expression or updated by the assignment, for example:

```
int fred, *ptr1, *ptr2;

ptr2 = 0;       /* sets the pointer to zero */
ptr1 = &fred;   /* sets the pointer to point at fred */
*ptr1 = 0;      /* sets the integer pointed to by ptr1 (fred) */
fred = *ptr1;   /* uses the integer pointed to by ptr1 */
ptr2 = ptr1     /* copies the pointer value */
```

3.5.2 Arrays

Arrays in QC are restricted to a single dimension (but see the next section). They are declared by following the array name with square brackets containing the array size (which must be a constant expression):

```
<type> <name> [ <constant expression> ]
```

For example:

```
int  array1 [ 10 * 100 ];
char buff [ 512 ];
```

If an array is declared in an **extern** directive, you leave out the array size, as it is specified in the other module.

To access an element in an array, the following syntax is used:

```
<name> [ <expression> ]
```

The first element in the array is accessed with an offset of zero, and if an array has N elements the last element is accessed using offset N-1.

An array name on its own stands for the address of the (first element of the) array. It can be used in expressions as the address of the array (eg it can be assigned to a pointer) but it cannot be altered itself.

Note that pointer syntax can also be used for addressing arrays: the following two expressions are equivalent:

```
array [ offset ];

* ( array + offset );
```

3.5.3 Simulating Multidimensional Arrays

Although QC will not create multidimensional arrays, it is possible to set up an array of pointers (sometimes known as an Iliffe Vector) which allows a single dimensional array to be accessed as though it is two dimensional.

An array of pointers is declared by putting both an asterisk and brackets on the array name, for example:

```
int  * index [16],  /* array of pointers to integers */
     table [256];   /* array of integers */
```

If the elements of index are set to point to every 16th element of table, we have simulated a 16 by 16 array. The pointers could be set up as follows:

```
int i;
for ( i=0 ; i < 16 ; ++i )
     index [ i ] = &table[ 16*i ];
```

The array can then be accessed as follows:

```
index[a][b];
```

index[a] takes one of the pointers in **index**, and gives a pointer into array **table**. The second array access gives us one of the elements in that part of array **table**.

3.6 Preprocessor Commands

These preprocessor commands effectively manipulate the C source before it is compiled. The preprocessor can be considered as an extra pass to the compiler, but the changes are actually made as the compiler reads the text.

All of these preprocessor directives should be used on a line on their own.

3.6.1 The Include Directive

The include directive has two forms (the angle brackets are part of the syntax here):

```
#include "filename"

#include <filename>
```

No spaces are allowed within the quotes or brackets.

This directive allows a file to be included in the compilation at this point. It can be used to make exactly the same set of definitions and directives in each of a set of modules.

Both forms of the directive as shown above are accepted by the compiler, but they are treated the same. (The two different forms indicate to some C compilers to search for the file in different directories.)

The filename specified in this directive should be the complete QDOS name of the file to be included, or if compiler option -D is being used (Section 2) the device or directory prefix can be omitted. The filename can be specified in upper or lower case.

Included files cannot be nested: if FILEA includes FILEB, and FILEB includes FILEC, then at the end of FILEC the rest of FILEB is skipped, and the compiler continues reading FILEA.

This directive can only be used at the top level of a program, ie amongst function declarations and global variable declarations. It cannot be used within a function.

3.6.2 Macro Substitution

#define <macroname> <macrotext>

This directive defines a macro which then stands for the arbitrary text on the rest of the line. When the macroname is encountered later in the module, the compiler replaces it by the macro text.

Macro names must obey the same rules as variable names, but it is conventional to use uppercase for macro names.

Macro names are commonly used to give names to constants, but they can be used to replace anything.

Macros can only be defined outside of functions, but macro substitution occurs anywhere in the rest of the program. (Except in strings and character constants.)

Note that macro substiution is performed on macro definition lines, so macros can be defined in terms of earlier macros.

Macros with parameters are not implemented in QC.

3.6.3 Conditional Compilation

#ifdef <macroname>

#ifndef <macroname>

#else

#endif

These directives allow the compiler to compile different sequences of code according to whether the macro is defined or not defined. Different versions of the same program can then be compiled by changing **#define** directives at the top of the program.

Conditional compilation directives can occur anywhere in a program.

3.6.4 Assembler Code

#asm

#endasm

These directives allow assembler code to be included in a program. They can be used outside of functions, or within a function anywhere where a statement is allowed. All 68000 instructions are accepted. For details of assembler syntax, see the Sinclair Macro Assembler manual. See section 6 for information on accessing C data from assembler code.

3.6.5 Listing Control Directives

```
#nolist

#list

#page
```

These directives can be used to control the layout and contents of the compiler listing. You might like to turn off listing for an included file containing a lot of macro definitions, or maybe start a new listing page for each function in a program.

4. QC Standard I/O Runtime Library

4.1 Introduction

The description of each function in this section starts with the function declaration, in bold, which defines the order and types of the parameters.

Programs which use the runtime library should include the standard header file **STDIO.H** which defines symbols for file descriptors and values returned by the library routines. For example:

```
#include "MDV2_STDIO.H"
#include <FLP1_STDIO.H>
```

If compiler option -D is used (see section 2) it is possible to leave out the device or directory name from filenames used in include statements, eg:

```
#include <stdio.h>
```

The definitions in STDIO.H include:

EOF = -1 returned by read routines when they reach the end of file

ERR = -2 which is returned by some routines when an error is detected.

NULL= 0 used as a null pointer value.
It is returned by some routines as a success code and by others as an error code.

YES = 1
NO = 0 These values are returned by some routines eg **feof**

4.2 Standard Input and Output

The following library routines can be called by programs which use only the standard input and standard output channels a character at a time. By default, the keyboard is read as the standard input, and standard output is sent to the screen.

See section 7 for information on I/O redirection.

Note that input and output are buffered by the QC library: if you call **putchar** to write a character to the screen, it will not appear immediately. Buffers are flushed at the end of each line, but see routine **fflush** (4.3.9).

The screen window used for standard I/O will not be cleared by tl library until it is written to. This means that programs which have redirected channels may not have any visible screen window. (Tagged windows are always displayed - see section 2.6)

4.2.1 getchar ()

Returns a character read from the standard input channel, or the value **EOF** at the end of file.

4.2.2 putchar (c) int c;

Writes the character to the standard output channel. The value **c** is returned if the write completed sucessfully, or **EOF** is returned if an error occurred.

4.2.3 puts (str) char *str;

This writes the string to the standard output, up to but excluding the null byte at the end. It then writes a newline to the standard output.

4.3 File Input and Output

These routines allow access to any file or device. In these calls, the parameter **fd** is a "file descriptor" which can be the value returned by one of the open calls (it should have the type pointer to int) or one of standard file descriptors:

stdin	for standard input
stdout	for standard output
stderr	for standard error output

Data written to **stderr** is always sent to the screen - it cannot be redirected like **stdout**. It is also permissible to read from **stderr**, in which case the keyboard will be read, regardless of **stdin**.

These standard files can be used without calling **fopen**.

4.3.1 fopen (name, mode)

```
char *name, *mode;
```

This opens a channel to the named file. It returns a file descriptor on success, or the value 0 if it fails. **mode** can be one of the following strings:

"r" open for read (the file must already exist)

"w" open for write (in which case the file is deleted first if it already exists, and then re-created)

"a" open for append (the file is created if it does not already exist, and is positioned to the end of file if it does exist. This cannot be used for output to devices such as the screen.)

"d" open a directory: see section 5.4.11

If this routine is called to open a channel to "CON_" or "SCR_" (ie without any window specification) then the same QDOS channel will be used as for **stderr**. If you want a separate window, put some window specification in the filename.

4.3.2 freopen (name, mode, fd)

```
char *name, *mode;
int *fd;
```

This can be used to close a channel then reopen it to another file. It returns **fd** on success, and zero on failure.

4.3.3 fclose (fd) int *fd;

This closes the channel and flushes any data still in memory to the operating system. Zero is returned for success and a nonzero value for error.

4.3.4 **getc (fd) int *fd;**
fgetc (fd) int *fd;

These routines are equivalent. They return a character read from the specified input channel, or **EOF** at end of file or if an error occurs.

4.3.5 **ungetc (c, fd) char c; int *fd;**

This function can be used to backup one place in an input file. It does not actually reposition the file, so it can be used on sequential devices. The next time the file is read the value will be returned. The value **c** is returned by **ungetc** unless previously saved character is in the buffer or if **c** has the value **EOF**, in which case **ERR** is returned.

4.3.6 **fgets (str, size, fd)**
char *str;
int size, *fd;

This routine is called to read a line from a file. **str** is a pointer to a buffer and **size** is the maximum number of bytes which may be read. Input is terminated by a newline character. A null byte is put in the buffer after the newline.

If the line is too big for the buffer, **size-1** data bytes are put in the buffer and null byte is put at the end.

The routine returns **str** on success or **NULL** on failure.

4.3.7 **putc (c, fd) char c; int *fd;**
fputc (c, fd) char c; int *fd;

These routines are equivalent. They write the character **c** to the file specified by **fd**. The value **c** is returned unless an error occurs in which case **EOF** is returned.

4.3.8 **fputs (str, fd) char *str; int *fd;**

This writes the string out to the file, up to but not including the terminating null byte. No newline is appended at the end.

4.3.9 **fflush (fd) int *fd;**

This routine is called to write data out to a file or device which has been buffered by the QC runtime library. It can be used to force a partial line to be written out to the screen: data to the screen is flushed automatically at the end of each line. This routine is called by **fclose.**

4.3.10 **isatty (fd) int *fd;**

This routine returns **YES** if the channel has been opened to a serial device (eg the screen or a comms line) or **NO** if the device supports random access (eg MDV or floppy disks).

4.3.11 **iscons (fd) int *fd;**

This routine returns **YES** if the channel is connected to the screen or keyboard otherwise it returns **NO.**

4.3.12 **delete (name) char *name**
unlink (name) char *name

These routines are equivalent. They delete the file specified by the filename, and return **NULL** on success or **EOF** on failure.

4.3.13 **feof (fd) int *fd;**

This routine returns **YES** if the fd has reached the end of file, else **NO.**

4.3.14 **ferror (fd) int *fd;**

This routine returns the system status code associated with the last system call for **fd.**

ferror (0) is a special case: it returns the system status code associated with the last file open call.

4.3.15 **clearerr (fd) int *fd;**

This clears any error status associated with the file fd.

4.4 Random Access I/O

The following routines can be used to move around files. A file position is a positive integer value, giving the position in bytes from the start of file.

4.4.1 rewind (fd) int *fd;

This routine repositions a file to the beginning. It returns NULL on success, **EOF** on error. It is equivalent to lseek(fd,0,0) (see below).

4.4.2 getpos (fd) int *fd;

This routine returns the current file position, or **EOF** if the channel does not support random access.

4.4.3 lseek (fd, offset, from) int *fd, offset, from;

This routine positions the file to a position **offset** bytes from the place specified by **from**:

```
from == 0      position relative to start of file
from == 1      position relative to the current position
from == 2      position relative to the end of file
```

The routine returns **NULL** on success or **EOF** on error.

4.5 Formatted I/O

4.5.1 printf (str, arg1, arg2, ...) char *str;

This function writes a formatted character string to the standard output channel. **str** is a "control string" which contains ordinary characters and conversion specifications. The ordinary characters are written out unchanged. Each conversion specification indicates how the corresponding **arg** is to be converted before output.

The function returns as its result the number of characters written.

The conversion specifications start with the character % and end with a letter. Between these characters can be optional fields giving extra formatting information:

a minus sign	indicates left-justify within field
a decimal number	gives field width: if specified with a leading zero, the field will be padded with zeros.
a decimal fraction	number of characters to take from string

The terminating letter indicates the type of conversion:

b	unsigned integer,	convert to binary
c	character	
d	signed integer,	convert to (signed) decimal
o	unsigned integer,	convert to octal
s	string address	
u	unsigned integer,	convert to (unsigned) decimal
x	unsigned integer,	convert to hexadecimal

If an invalid character is encountered, it is written out as a text character, so %% in the format string is written out as %

See the next page for some examples of printf.

4.5.2 fprintf (fd, str, arg1, arg2,...) int *fd; char *str;

This routine is just like **printf** except that the first parameter indicates the channel to be written to.

PRINTF EXAMPLE

Vertical bars have been included in the format string examples to show the effect of spacing and field width in output using **printf**. They are not mandatory.

if **name** is the string "fred" and **score** is an integer 67, then

```
printf ( "%s's score is %d%%\n", name, score );
```

will print:

```
fred's score is 67%
```

format string	parameter(s)	printout
|%d|	1234	|1234|
|%6d|	1234	| 1234|
|%-6d|	1234	|1234 |
|%06d|	1234	|001234|
|%d|	-1	|-1|
|%u|	-1	|4294967295|
|%x|	-1	|ffffffff|
|%b|	1234	|10011010010|
|%06o|	1234	|002322|
|%04x|	1234	|04d2|
|%c|	'A'	|A|
|%9c|	'A'	| A|
|%2x|	'A'	|41|
|%s|	"computer"	|computer|
|%5s|	"computer"	|computer|
|%12s|	"computer"	| computer|
|%-12s|	"computer"	|computer |
|%12.4s|	"computer"	| comp|
|%-9s %02d/%02d/%02d|	"Thursday",14,2,85	|Thursday 14/02/85|

4.5.3 **scanf (str, arg1, arg2,...) char *str;**

This routine performs an inverse function to printf - it reads text from the standard input and interprets it according to the conversion specifications in **str**.

The control string **str** may contain only conversion specifications and white space (which is ignored).

All the **args** passed to scanf should be **addresses** of variables where the results are placed. (Or, for strings, a buffer pointer.)

The function returns as its value the number of fields processed - it will stop early if input data does not match the conversion specification. If no fields have been read and it reaches end of file, it will return **EOF**.

Conversion specifications are similar to printf, but between the % and the letter you can only put an asterisk (which indicates skip this field) and/or a decimal number, giving the field width.

A field is normally a sequence of printing characters, terminated by a white-space character. A field is also terminated when the field-width (if specified) has been reached. If the conversion specification is %c, a single character is read, without skipping whitespace.

The format specifications are:

b	binary integer
c	character
d	(signed) decimal number
o	octal integer
s	character string
u	unsigned decimal number
x	hexadecimal number

See the next page for examples of scanf.

4.5.4 **fscanf (fd, str, arg1, arg2, ...) int *fd; char *str;**

This routine works just like **scanf** but the first parameter indicates which channel to read from.

SCANF EXAMPLE

Consider this statement:

```
scanf ( " %s   %c   %c  %*s  %d   %3d  %d ",
          str, &c1, &c2,     &i1, &i2, &i3 );
```

If the input contains the following text:

```
abc defg   -12 345678 9
```

Then the variables will receive these values:

str:	"abc"	reads a string terminated by space
c1:	' '	reads the next char
c2:	'd'	reads the next char the next string is skipped: "efg"
i1:	-12	reads a number terminated by a non-digit
i2:	345	reads a 3-digit number
i3:	678	reads a number terminated by a non-digit

The next input call will read starting at the space after "345678".

As a further example, this statement:

```
num = scanf ( " %d %*c %d %*c %d *%c ", &i1, &i2, &i3 );
```

could be used to read numbers terminated with non-digits:

```
123, 456, 789,
```

as %d reads up to a non-digit and %*c skips one character.

4.6 Format Conversion Functions

4.6.1 atoi (str) char *str;

Convert (signed) decimal number in **str** to an integer. Leading whitespace is skipped. A sign character (+ or -) may optionally precede the first digit. Conversion stops on the first non-digit.

4.6.2 atoib (str, base) char *str; int base;

Convert (unsigned) number in specified base to an integer. Leading whitespace is skipped. Bases 2 to 16 are allowed.
Currently, this routine destroys the contents of the string.

4.6.3 itoa (num, str) int num; char *str;

Convert the number to decimal, left-justified in the string. If the number is negative, a minus sign is generated. The string is terminated by a null byte. 32-bit integers can require up to ten digits (or twelve bytes including sign & null).

4.6.4 itoab (num, str, base) int num; char *str; int base;

Convert the (unsigned) number to a string using the specified base. Bases 2 to 16 are allowed. The string is terminated by a null. Up to 33 bytes can be generated (for base=2).

4.6.5 dtoi (str, num) char *str; int *num;

Convert string from signed decimal, put the result in **num.** Returns the no. of digits in the number, or ERR if an error occurred (eg overflow or no valid digits). The routine does not accept leading spaces or a plus sign but will accept an optional leading minus sign.

4.6.6 otoi (str, num) char *str; int *num;

Convert string from unsigned octal, put the result in **num.** Returns no. of digits in the number, or ERR on overflow. It does not skip leading spaces.

4.6.7 utoi (str, num) char *str; int *num;

Convert string from unsigned decimal, put the result in **num.** Returns no. of digits, or ERR on overflow. It does not skip leading spaces.

4.6.8 **xtoi (str, num) char *str; int *num;**

Convert string from unsigned hex, put the result in **num.** Returns no of digits or ERR on overflow. Upper and lowercase letters are accepted. It does not skip leading spaces.

4.6.9 **itod (num, str, size) int num, size; char *str;**

Convert integer **num** to signed decimal, right-justified in **str.**
If **size** > 0, **size**-1 bytes are put in **str** followed by a null byte.
If **size** = 0, it searches for a null byte in **str.**
If **size** < 0, -**size** bytes are put in **str,** with no null terminator.
Returns **str.**

4.6.10 **itoo (num, str, size) int num, size; char *str;**

Convert integer **num** to octal, otherwise like **itod.**

4.6.11 **itou (num, str, size) int num, size; char *str;**

Convert integer **num** to unsigned decimal, otherwise like **itod.**

4.6.12 **itox (num, str, size) int num, size; char *str;**

Convert integer **num** to hex, otherwise like **itod.** Digits A to F are written in uppercase.

4.7 String and Character Handling Functions

4.7.1 left (str) char *str

This routine left-adjusts the string. Starting at the first non-blank character, the bytes are moved to the start of the string up to and including the terminating null byte.

4.7.2 strcat (dest, sour) char *dest, *sour;

The string **sour** is appended to the string **dest** which must be big enough: it is up to the programmer to ensure this.

4.7.3 strncat (dest, sour, n) char *dest, *source; int n;

This is like **strcat** except that **n** limits the number of characters which can be taken from **sour**. It does not fill the buffer with spaces like **strncpy**.

4.7.4 strcmp (str1, str2) char *str1, *str2;

This returns an integer value less than, equal to, or greater than zero, depending on whether the string **str1** is less than, equal to, or greater than **str2**.

Comparison is based on the ASCII values of the characters in the strings, but the result is undefined for strings containing bytes in the range 128 to 255 (i.e. with the top bit set) - it will give the right answer for string equality, but the inequality result is undefined.

If the strings match up to the length of the shortest one, that string is considered to be the lesser of the two.

4.7.5 strncmp (str1, str2, n) char *str1, *str2; int n;

This function is like **strcmp** but a maximum of **n** characters are compared.

4.7.6 strcpy (dest, sour) char *dest, *sour;

The string **sour** is copied to **dest**. Beware of overlapping strings: a string can be shuffled to the left using this routine, but not to the right.

4.7.7 **strncpy (dest, sour, n) char *dest, *sour; int n;**

The string is copied like strcpy, but if it is too short it is padded to **n** chars with spaces and if it is too long it is truncated to **n** characters. A null byte is then put at the end. Overlapping strings should be avoided.

4.7.8 **strlen (str) char *str;**

This function returns the length of the string (excluding the null terminator byte).

4.7.9 **strchr (str, c) char *str, c;**

This searches for the first occurrence of the character **c** in the string, and returns a pointer to it. It returns NULL if the character is not found.

4.7.10 **strrchr (str, c) char *str, c;**

This searches for the last occurrence of the character **c** in the string, and returns a pointer to it. It returns NULL if the character is not found.

4.7.11 **reverse (str) char *str;**

This routine reverses the order of the characters in the string.

4.7.12 **lexcmp (str1, str2) char *str1, *str2;**

This routine compares strings like **strcmp** except it uses **lexorder** to compare characters, rather than using ASCII sequence.

4.7.13 **lexorder (c1, c2) char c1, c2;**

This routine compares two characters using dictionary order rather than ASCII order:

Control codes come first

Other non-alpha chars precede all alphabetic chars

Uppercase letters sort immediately before the corresponding lowercase letters

DEL (copyright) comes last

Top-bit-set characters are undefined.

4.8 Character Classification Functions

These routines return **YES** or **NO** according to whether the char belongs to the specified class of characters. They do not currently recognise any of the Sinclair extended character set for foreign alphabets.

4.8.1 **isalnum (c) char c;**
alphanumeric characters: 0-9 A-Z a-z

4.8.2 **isalpha (c) char c;**
alphabetic characters: A-Z a-z

4.8.3 **isascii (c) char c;**
ASCII characters: 0-127

4.8.4 **iscntrl (c) char c;**
ASCII control codes: 0-31

4.8.5 **isdigit (c) char c;**
digits: 0-9

4.8.6 **isgraph (c) char c;**
graphic characters: 33-127 (excludes space)

4.8.7 **islower (c) char c;**
lowercase letters: a-z

4.8.8 **isprint (c) char c;**
printing characters: 32-127 (includes space)

4.8.9 **ispunct (c) char c;**
punctuation characters:
ASCII chars excluding control codes and alphanumerics

4.8.10 **isspace (c) char c;**
whitespace characters: space and control codes

4.8.11 **isupper (c) char c;**
uppercase letters: A-Z

4.8.12 **isxdigit (c) char c;**
hex digits: 0-9 A-F a-f

4.9 Character Conversion Functions

4.9.1 toascii (c) char c;

Converts c to ASCII (it leaves it unchanged).

4.9.2 tolower (c) char c;

If c is an uppercase letter, return the lowercase equivalent, else return c.

4.9.3 toupper (c) char c;

If c is a lowercase letter, return the uppercase equivalent, else return c.

4.10 Other System Facilities

These are assorted system facilities which are often provided in a similar form in other C implementations. System facilities which are specific to QDOS are listed in section 5.

4.10.1 abs (n) int n;

This returns the absolute value of the integer n.

4.10.2 sign (n) int n;

This returns -1, 0 or +1 according to the sign of n.

4.10.3 fread (buff, size, count, fd) char *buff; int size, count, *fd;

This reads from file **fd** into **buff**, **count** items **size** bytes long. It performs a binary transfer. It returns the actual number of items read: use feof() or ferror() to determine if at end of file or if an error occurred.

4.10.4 fwrite (buff, size, count, fd) char *buff; int size, count, *fd;

This writes **count** items **size** bytes long from **buff** to file **fd.** It performs a binary transfer. It returns the actual number of items written: use feof() or ferror() to determine if at end of file or if an error occurred.

4.10.5 read (fd, buff, count) char *buff; int count, *fd;

This reads **count** bytes from file **fd** into **buff.** It performs a binary transfer. It returns the actual number of bytes read: use feof() or ferror() to determine if at end of file or if an error occurred.

4.10.6 write (fd, buff, count) char *buff; int count, *fd;

This writes **count** bytes from **buff** to file **fd.** It performs a binary transfer. It returns the actual number of bytes written: use feof() or ferror() to determine if at end of file or if an error occurred.

4.10.7 calloc (count, size) int count, size;

This allocates **count** * **size** bytes of memory, and initialises them to zero. It returns a pointer to the memory, or zero if memory is exhausted.

4.10.8 **malloc (count) int count;**

This allocates **count** bytes of uninitialised memory. It returns a pointer to the memory or zero if out of memory.

4.10.9 **avail (abort) int abort;**

This routine returns the amount of free space left between the program and the stack. If **abort** is nonzero and the stack has overwritten the program, the program will be aborted.

4.10.10 **free (pointer) char *pointer;**
cfree (pointer) char *pointer;

These routines are equivalent. They return memory to the heap which was grabbed by calls to calloc or malloc. Any record may be released back to the heap at any time: it is not necessary to release records in the reverse order of allocation.

The pointer must be a pointer as returned from calloc or malloc otherwise the system heap may be corrupted: a pointer into a record allocated from the heap will not work.

4.10.11 **getarg (n, str, size, argc, argv)**
char *str; int n, size, argc, *argv;

Extract the **n**th argument from the program's parameter string and copy it into **str** (maximum size **size**). See section 7 for more info on program parameter strings.

4.10.12 **poll (pause) int pause;**

This routine looks to see if there are any keystrokes pending for the program. If **pause** is zero, any character is returned to the caller (or zero if there are no chars waiting).

If **pause** is nonzero, and the character is control-S, then the program is suspended until the next character is entered on the keyboard (the character will not be read by the program).

4.10.13 **abort (errcode) int errcode;**
exit (errcode) int errcode;

These routines are equivalent. They close all open files and return to the system. Errcode should be zero (to indicate success) or a QDOS error code.

4.10.14 **ccargc()**

This function is called to determine the number of parameters passed to a user function. See section 3.4.8

5 Extra QDOS Library Routines

The routines defined in this section are included to allow access to the facilities of the QDOS operating system. They are unique to QC and are non-portable.

5.1 Interfacing with QDOS

trap1(regpointer)
trap2(regpointer)
trap3(regpointer)

These three routines allow direct access to almost all of the QDOS facilities implemented as traps. See the "QDOS Software Developers Guide", available from Sinclair Research Ltd, for more information on how to use the traps.

Regpointer is a pointer to an array of 8 integers, arranged as

D0,D1,D2,D3,A0,A1,A2,A3

all of which are updated with result of the trap.

To access the QDOS channel associated with a QC file **fd** (defined as **int *fd**), use ***fd**. As an example, here are some routines for unbuffered I/O:

```
/*   INKEY
** This routine provides a similar facility to SuperBasic's
** INKEY function: it reads the keyboard directly or
** returns 0 if no key was pressed during the timeout period.
*/

inkey ( timeout )
     int timeout;
{
     int regs[8];

     regs[0] = 1;         /* IO.FBYTE */
     regs[3] = timeout;
     regs[4] = *stderr;   /* A0 = channel to CON_ */

     trap3( regs );

     if ( regs[0] == 0 )           /* if no error, */
          return ( regs[1] );      /* return key code */
     else
          return ( 0 );
}
```

```
/*   OUTSCR
** This is a companion routine to INKEY.
** It is used to write characters to the screen avoiding
** QC's buffers.
*/

outscr ( ch )
     int ch;
{
     int regs[8];

     regs[0] = 5;          /* IO.SBYTE */
     regs[1] = ch;         /* D1 = char to write */
     regs[3] = -1;
     regs[4] = *stderr;    /* A0 = channel to CON_ */

     trap3( regs );

     return ( regs[0] ); /* return status code, if any */
}
```

5.2 Screen and Window functions

All the functions in this section call TRAP 3 to update the screen. They return QDOS status codes.

In order to change text colour or style within a line, you must call fflush() to write out the text before the screen driver call.

5.2.1 selwindow(fd)

```
int *fd;
```

This routine is used to select which window to is to be used for graphics and screen driver routines. The default is the window connected to **stderr**.

5.2.2 getwindow(flag,pointer)

```
int flag, pointer[];
```

This routine is used to enquire the window size and cursor position. Flag is 0 to get the result in pixel coordinates, or nonzero to get the result in character coordinates.

Pointer specifies where the results are to be put: it should be an array of four integers (or a pointer into a bigger array).

```
pointer[0]    gets window width
pointer[1]    gets height
pointer[2]    gets cursor x position (from left)
pointer[3]    gets cursor y position (from top)
```

5.2.3 border(size,colour)

```
int size, colour;
```

This routine is used to update the window border. Size is the required border width and colour is the required colour.

5.2.4 window(width, height, x, y)

```
int width, height, x, y;
```

Change window definition. width and height specify the new window size. x and y specify the window position. The cursor is reset to the top left corner of the window (position 0,0).

5.2.5 cursen(switch)

```
int switch;
```

This routine turns the cursor on or off, according to the value of switch: 0 means off, nonzero means on.

5.2.6 **at(line,col)**
```
int line, col;
```

This routine positions the cursor at the specified line and character position within the window.

5.2.7 **tab(col)**
```
int col;
```

This routine positions the cursor at the specified column number.

5.2.8 **nextline()**

This routine moves the cursor to the start of the next line.

5.2.9 **curleft()**

This routine moves the cursor left one space.

5.2.10 **curright()**

This routine moves the cursor right one space.

5.2.11 **curup()**

This moves the cursor to the next row up.

5.2.12 **curdown()**

This moves the cursor to the next row down.

5.2.13 **cursor(xpos,ypos)**
```
int xpos, ypos;
```

This moves the cursor to the specified (pixel addressed) position.

5.2.14 **scroll(distance,part)**
```
int distance, part;
```

The window (or part of it) is scrolled vertically by the specified amount. A positive distance moves the text down on the screen. Part is interpreted as follows:

```
part=0    Scroll window
part=1    Scroll top of window (lines above cursor line)
part=2    Scroll bottom of window (lines below cursor line)
```

5.2.15 **pan(distance,part)**
```
int distance, part;
```

The window (or part of it) is scrolled horizontally by the specified amount. A positive distance scrolls the text to the right. Part is interpreted as follows:

```
part=0    Pan whole window
part=3    Pan whole cursor line
part=4    Pan cursor line (from cursor to end of line)
```

5.2.16 **cls(part)**
```
int part;
```

The window (or part of it) is cleared to the current paper colour. Part is interpreted as follows:

```
part=0    Clear window
part=1    Clear top of window (above cursor line)
part=2    Clear bottom of window (below cursor line)
part=3    Clear whole cursor line
part=4    Clear cursor line (from cursor to end of line)
```

5.2.17 **fount(fount1,fount2)**
```
char *fount1, *fount2;
```

This routine is used to select the character fount for the window. A zero fount address selects the default fount. See QDOS documentation for details of the fount data structure. If a character to be written is not defined in fount1 then fount2 will be checked.

5.2.18 **recol(table)**
```
char *table;
```

This routine recolours a window. Table is an array of eight bytes (chars) giving the new colour (0-7) for each of the colours currently displayed on the screen. (Order is 0=black, 1=blue, 2=red, 3=magenta, 4=green, 5=cyan, 6=yellow, 7=white.)

5.2.19 **paper(colour)**
```
int colour;
```

Set paper colour. The paper colour is used when clearing the screen, and is also used to fill the space left behind by scroll and pan.

5.2.20 **strip(colour) int colour;**

Set strip colour. The strip colour is used as the background colour when text is written to the screen (Unless it is being written as transparent text). The strip colour is often set to the same as the paper colour.

5.2.21 **ink(colour) int colour;**

Set ink colour. This is used for text and graphics (except when writing in XOR mode).

5.2.22 **flash(switch) int switch;**

Set or unset flash mode. Nonzero switch turns on flashing, zero switch turns it off. Flashing only works if the machine is in 8 colour mode.

5.2.23 **under(switch) int switch;**

Set underlining on or off according to the value of switch.

5.2.24 **over(switch) int switch;**

Set writing and plotting mode:

switch = -1	XOR mode
switch = 0	character background is strip colour
switch = +1	transparent mode

5.2.25 **csize(width,height) int width, height;**

Character size and spacing:

width = 0	6 pixels spacing
1	6 pixels on 8 pixels spacing
2	12 pixels wide
3	12 pixels on 16 pixels spacing

height = 0 for single height, 1 for double height

5.2.26 **block(width,height,xpos,ypos,colour)**
int width, height, xpos, ypos, colour;

Fill a rectangle (specified in pixel coordinates) in the specified colour.

5.3 Graphics Routines

In these graphics routines, all coordinates are in graphics units which are scaled to fit the window. Angles are specified in hundredths of radians as QC does not support floating point.

The ellipse eccentricity should also be passed as one hundred times the required number.

The graphics are drawn in the window selected by the **selwindow** routine (5.2.1) using the ink colour selected by **ink** (5.2.21).

Note that graphics coordinates have the origin at the bottom left, unlike pixel coordinates which have the origin at the top left.

5.3.1 point(x,y)

```
int x, y;
```

5.3.2 line(x1,y1,x2,y2)

```
int x1, y1, x2, y2;
```

5.3.3 arc(x1,y1,x2,y2,angle)

```
int x1, y1, x2, y2, angle;
```

The arc is drawn between the two end points. The sense of the arc depends on the sign of the angle: if +ve, the curve is drawn in an anticlockwise direction.

5.3.4 circle(xpos, ypos, radius)

```
int xpos, ypos, radius;
```

This routine draws a circle, which is a special case of the ellipse routine. A circle is equivalent to an ellipse with eccentricity of 1:

```
circle(xpos, ypos, radius)
     int xpos, ypos, radius;
{
     ellipse ( xpos, ypos, radius, 100, 0 );
}
```

5.3.5 ellipse(xpos, ypos, radius, eccentricity, angle)

```
int xpos, ypos, radius, eccentricity, angle;
```

The **radius** specifies one of the ellipse radii. The other radius is specified by **radius** times eccentricity. Note that the **eccentricity** parameter is scaled by 100, so a value of 100 produces a circle.

5.3.6 **scale(scalefactor,xorg,yorg)**
int scalefactor, xorg, yorg;

The scale is adjusted so that the height of the window is equivalent to **scalefactor** graphics units. **xorg** and **yorg** give the internal graphics coordinates of the bottom left hand corner of the window. The default scale is 100, and the default origin is 0,0.

5.3.7 **gcursor(xorg,yorg,right,down)**
int xorg, yorg, right, down;

The cursor position is taken from **xorg** and **yorg** in graphics coordinates plus an offset **right** and **down** in pixel coordinates.

5.3.8 **fill(switch)**
int switch;

Area fill is turned on or off according to the value of **switch.**

5.4 Other QDOS Facilities

5.4.1 delay (ticks) int ticks;

Delay for a short period. A tick is one fiftieth of a second.

5.4.2 adate(seconds) int seconds;

Adjust the clock forwards or backwards by a number of seconds. Returns the clock value in seconds from the first of January 1961.

5.4.3 qdosdate()

Returns the clock value.

5.4.4 date(clock,datevec)
int clock; int datevec[];

Converts **clock** to a formatted date: datevec should be a pointer to an array of 7 integers arranged as:

```
datevec[0] = year
datevec[1] = month
datevec[2] = day in month
datevec[3] = weekday (0=Sunday, 1=Monday, .. 6=Saturday)
datevec[4] = hour
datevec[5] = minute
datevec[6] = second
```

5.4.5 sdate(value)
int value;

Set the clock.

5.4.6 beep(duration,pitch)
int duration, pitch;

This routine provides a simple interface to the QL sound generator. Duration is the period in units of 72 microseconds. (14000 is therefore about one second.) The lower the pitch number, the higher the tone produced.

An even simpler way to drive the QL sound generator is provided in the main library: writing a control-G to the screen will generate a short beep.

Note that this routine (and the one below) return immediately. To wait for the beep to finish, call **delay**. The time to delay is approx **duration**/256 ticks.

5.4.7 **warble(duration,pitch1,pitch2,interval,step,wrap,fuzz,rand)**

```
int  duration, pitch1, pitch2, interval,
     step, wrap, fuzz, rand;
```

This routine gives the user full control over the sound generator. See the SuperBasic manual for more information on the various parameters.

5.4.8 **keyrow(row)**

```
int row;
```

This reads the QL keyboard directly. It can be used to tell when two or more keys are pressed. See QDOS and SuperBasic documentation for more details. Bits set in the result indicate which keys in that row are held down.

5.4.9 **random()**

Returns a random integer.

5.4.10 **rnd(min,max)**

```
int min, max;
```

Returns a random integer in the specified range.

5.4.11 **readdir(fd, fname, dirinfo)**
int *fd, *dirinfo;
char *fname;

This routine is used to read the contents of a directory. **fd** should be the result of a call to **fopen** with option = "d". **fname** is a pointer to an array of characters big enough for a filename (37 characters is enough) and **dirinfo** is a pointer to an array of eight integers.

The routine returns zero normally, or EOF at the end of the directory. An example of the use of this routine:

```
main()
{    int dirinfo[8]; char filename[40]; int *fd;
     fd = fopen ( "mdv1_", "d" );
     while ( readdir ( fd, filename, dirinfo )==0 )
     {    printf ( ....
     }
}
```

The information in the directory entry is put into the array as follows:

```
dirinfo[0]     file size
dirinfo[1]     key
dirinfo[2]     file type
dirinfo[3]     type dependent information
dirinfo[4]     "    "    "    "    "
dirinfo[5]     date of last update
dirinfo[6]     date of last access
dirinfo[7]     date of last backup
```

Read the QDOS documentation for more information on the information held in a directory. Note that not all of these information fields are implemented on all device drivers.

5.4.12 **exec (progname, optstr, flag)**
char *progname, *optstr; int flag;

This routine allows you to run another program. **progname** is the name of the program to run and **optstr** is passed to the program as a parameter string. If **flag** is zero, the routine returns immediately after the subprogram was started. If nonzero, the routine waits for the subprogram to finish.

A QDOS status code is returned by this routine. If the routine waits for the subprogram to finish, the status code returned by that program is returned by this routine.

6 Interfacing with Assembler Code

6.1 Register Usage

The 68000 registers are used by QC as follows:

D0,D1 Hold temporary values for arithmetic calculations.

D2 The number of parameters passed to a routine.

D3,D4,D5 Not used.

D6 Always holds the value 1.

D7 Always holds the value 0.

A0 The "primary register", holds the current value of the expression being evaluated. Holds the result of a function when returned to caller.

A1 The "secondary register", used when evaluating expressions and often holds the address of a variable being updated in an assignment.

A2,A3 Not used

A4 Points to the "middle" of the program. All functions are accessed via an offset from pointer A4.

A5 Points to the base of the global variables. All global variables are accessed using an offset from this register.

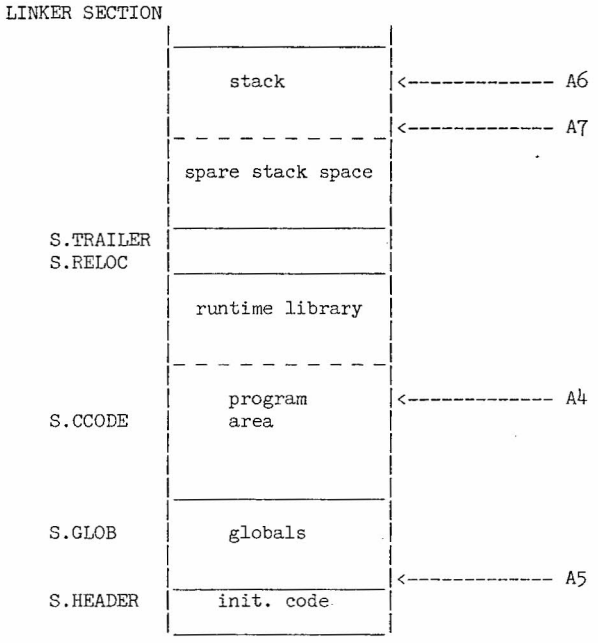
A6 The "stack frame pointer" points at the current function's local variables on the stack.

A7 The stack pointer points at the "top" of stack. (On the 68000, the stack grows down from high memory.)

If you write assembly code to be interfaced with QC, you are free to use any register, but the following registers must be restored when you return to QC code: D6, D7, A4, A5, A6, A7

6.2 The Memory Map of a QC Program

High addresses are at the top of the diagram.



Sections S.HEADER, S.TRAILER and S.RELOC should not be used by the assembler programmer: all code should be put into section S.CCODE along with code generated by the compiler. The location pointed to by A4 is referenced by the symbol M$ so functions can be referenced as follows:

```
JSR       func-M$(A4)
```

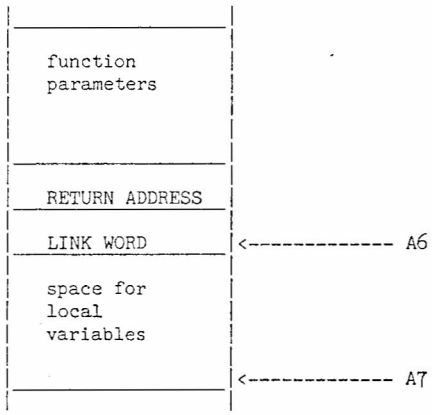
Using this mechanism, any function within 32k of A4 can be accessed in a position-independent manner, allowing QC programs of up to around 64k in size. (M$ is actually defined to be 32k from the base of the program.)

Global variables can be declared in assembler code by switching to section S.GLOB and using DC.L or DC.B to declare variables and their initial values. The location pointed to by A5 is referenced by the symbol G$ so other global variables can be referenced as follows:

```
MOVE.L    globname-G$(A5),D0
MOVE.B    D1,globchar-G$(A5)
```

6.3 QC Stack Structure

QC uses the 68000 instructions LINK and UNLK to maintain a stack frame structure: on entry to a function, the function parameters have been pushed onto the stack by the caller. The function uses a LINK instruction to reserve space for local variables declared at the outer level of the function. The stack then looks like this (high addresses at the top of the diagram):



The function parameters are accessed using positive offsets from A6, and the local variables are accessed using negative offsets. If a block within a function declares local variables, another LINK instruction is not generated: the stack pointer (A7) is just modified.

At the end of the function, UNLK is used to restore the stack to its state before the LINK, then RTS to return to the caller. Parameters are removed from the stack by the caller.

All parameters are passed to functions as **ints**, i.e. 32-bit longwords. They are pushed onto the stack in the order they are declared, so the first parameter is furthest from A6 and the last parameter is at address 8(A6).

Amongst the local variables, int and pointer variables occupy 4 bytes and are word aligned. Short takes 2 bytes. Character variables occupy just one byte - gaps may be left between variables if you have a mixture of char and int.

If in doubt about how to access local variables in an assembly code insert in QC code, look at the output from the compiler.

The simplest way to use assembly code inserts is to use the fact that the compiler leaves the result of an expression in the register A0 which can then be used in your assembler code. If assembly code appears at the end of a function, the value you leave in A0 will be the result of the function.

6.4 Example of a Code Insert

This example is taken from the graphics module in the library. It uses the QDOS floating point routines to manipulate real numbers on a floating point stack. **fp_tos** is a pointer into the array **fp_stack**.

```
char fp_stack[300], *fp_tos;

/*   FP_DIVIDE
** Divide the "next on stack" by the "top of stack"
*/

fp_divide()
{
     fp_tos;   /* get the floating point stack pointer into A0 */
#asm
     MOVE.L    A0,A1          get fp stack pointer in A1
     MOVE.L    A6,-(SP)       save C's A6
     SUB.L     A6,A6          set A6 to zero for QDOS
     MOVEQ     #$10,D0        QDOS code for divide
     MOVE.L    $11C,A2        get address of FP routine
     JSR       (A2)           call QDOS floating point
     MOVE.L    (SP)+,A6       restore C's A6
#endasm

     fp_tos += 6;   /* update our copy of stack pointer */
}
```

7 The Command Line and I/O Redirection

If you have the QL Toolkit, or if you start programs with the library routine **exec**, it is possible to pass information to a program when you execute it. You can also redirect the standard input and standard output channels.

7.1 Passing a Command Line to a Program

Using EX, EW or ET you can pass a text string to a QC program started from SuperBASIC. See the QL Toolkit documentation for full details. The command looks like this:

```
EX   <programname> ;     <string>
```

where <string> should be a SuperBASIC string expression.

The command line string will be parsed by the QC startup code, so that the words in the command line can be processed individually. (It looks for sequences of non-space characters separated by one or more spaces.)

7.2 Redirecting I/O Channels

If data filenames are included in the EX command line, the first datafile will be taken as the program's standard input, and the last one will be used as the standard output (but if pipes have been set up these will be used instead).

Examples:

```
EX   QC,MYFILE_C,MYFILE_ASM
```

This runs the compiler which reads from MYFILE_C on the default data device and writes to MYFILE_ASM on the program device. Note that filenames here can have the device name omitted, as it is appended by the toolkit, but the extensions must be defined explicitly.

```
EX   MYPROG_BIN,DATAFILE
```

This runs the program which reads from file DATAFILE, but standard output still goes to the screen.

```
EX   MYPROG_BIN,DATAFILE,AFILE     TO   ANOTHERPROG
```

In this case program MYPROG reads from DATAFILE, ignores and sends its output down a pipe to ANOTHERPROG which can be another QC program. (See the QL Toolkit documentation)

For compatibility with other operating systems, the UNIX-like convention using angle brackets can also be used for I/O redirection.

If the following forms appear in the EX option string they will override any other I/O redirection. No spaces are allowed between the angle bracket characters and the filenames here.

<filename	open the file as standard input
>filename	open the file as standard output
>>filename	open the file as standard output, but append the text to the end of the file.

Redirection specifications like these are not passed to the user program as parameters: see below.

7.3 Interpreting the Command Line Within a Program

To use the command line passed from SuperBASIC, the **main** function in the program should be declared as follows:

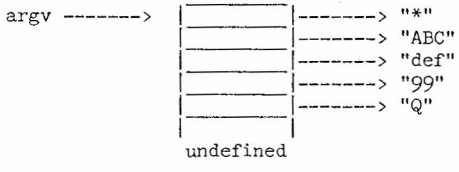
```
main ( argc, argv )
     int argc, *argv[];
```

argc is the number of arguments passed to the program (plus one for the program name), and **argv** is a pointer to an array of pointers to the argument strings.

If, for example, a program is started with the following command:

```
EX   PROG; " <MDV2_DATAFILE ABC  >>MDV1_FILENAME def 99   Q "
```

argc will have the value 5 and **argv** will point to an array like this:



Note that argv[0] points to a string containing an asterisk. This is set up as a pointer to the program name on operating systems which support this.

The library routine **getarg** can be used to access parameters in this data structure. It is defined as follows:

```
getarg ( n, str, size, argc, argv )
     char *str;
     int n, size, argc, *argv;
```

The routine copies the **n**th parameter into the string **str**, and it will be truncated to **size** bytes if it is too big. **argc** and **argv** should be passed as received from the operating system, eg:

```
main ( argc, argv )
     int argc, *argv;
{
     char buff[20];

     getarg ( 2, buff, 20, argc, argv );
}
```

getarg returns EOF and puts a null byte in **str** if it reaches the end of the argument list.

APPENDIX A

Compiler Error Messages

When the compiler detects errors in the code, it prints an error message on the screen and to the listing file (if a listing is being produced).

Error messages to the screen consist of the line in which the error was detected, followed by a line with a pointer indicating where in the line the error was detected followed by a message.

If option -A was selected, the compiler will bleep to draw your attention. If option -P was selected, the compiler will pause and wait for you to type ENTER before proceeding.

At the end of the compilation, the compiler will print a list of any undefined symbols on the screen and to the listing file. The list is in two parts: first any undeclared names which have been used as function names are listed: these are assumed to be external functions. This is followed by a list of undeclared names which have not been used as function names. These are assumed to be undefined globals, and are treated as an error.

The compiler error messages are listed below in alphabetical order together with explanations.

already defined

The symbol has already been used. You can create a local variable with the same name as a global variable or a local in another block, but you cannot have two global variables with the same name or two local variables in the same block with the same name.

bad label

The label name is invalid or missing.

can't subscript

You cannot subscript something which is not a pointer or an array.

cannot assign to pointer

Pointers cannot be initialised, except character pointers which can be initialised with a string constant.

cannot assign

An attempt was made to assign to something which is not an <lvalue>. (This error message is also produced if the increment or decrement operators are used on something which is not an <lvalue>.) An <lvalue> is something which can validly appear on the left side of an assignment: typically a variable or a subscripted array or a indirected pointer.

cannot initialise local arrays

Global arrays can be initialised, but local arrays cannot.

error opening file

An error occurred opening one of the files named in the command line. This is a fatal error.

expression too complicated

The compiler can handle most expressions with up to 12 levels of brackets. If you get this error, simplify the expression or break it into several smaller expressions and use some temporary variables.

failed to open include file

An error occurred when attempting to open an include file. The compiler will carry on after this error.

function body must be a compound statement

QC requires a function body to be a compound statement: other statement types are not allowed.

global symbol table overflow

There are too many global symbols for the compiler's symbol tables. There is room for around 200 global symbols. If you get this error, try breaking the program down into smaller and simpler modules.

illegal address

The address operator & was used on something which does not have an address.

illegal argument name

There is some syntax error in the function argument name (eg it is a reserved word).

illegal array size

Negative array sizes are illegal.

illegal function or declaration

This is a syntax error at the level of defining functions or global variables: the compiler cannot make sense of the declaration.

illegal symbol

The symbol name contains invalid characters or is a reserved word.

invalid expression

An expression term is invalid. Valid terms are constants, variables, or string constants. Label names and reserved words are invalid.

line too long

After the preprocessor has performed macro substitutions, the line has exceeded 128 characters. You should simplify the line or break it over several lines.

literal queue overflow

There are too many string constants in a function, or they are too long. The compiler saves the string constants until the end of the function. There is room for about 800 characters. If you get this error, break the function into smaller functions.

local symbol table overflow

There are too many local symbols in the function for the compiler's local symbol table. Break the function into smaller functions.

locals not allowed in switch

Local variables are not allowed in the block controlled by a **switch** statement.

locals not allowed with goto

Local variables are not allowed in a block if the function has any **goto** statements.

macro name table full

There are too many macro names for the compiler's macro symbol table. If you get this error, you should break the program into several smaller modules. There is room for around 100 macro names.

macro string queue full

The space reserved for macro definitions has overflowed. Break the program into smaller modules.

missing final closing bracket

End of file was reached while still inside a function.

missing token:

Some syntax element (eg a close bracket or a comma) was expected but not found. The error message shows which character was expected.

multiple defaults

Only one **default** label is allowed in a **switch** statement.

must assign to char pointer or array

Only char pointers or char arrays can be initialised with string constants. Simple char variables cannot be initialised in this way.

must be constant expression

A variable expression was used where a constant expression is required (eg when defining the size of an array or when initialising variables.)

must declare locals at start of block

Local variables must be declared before any statements within a block.

no apostrophe

A string constant does not have a terminating apostrophe or single quote character.

no final }

End of file was reached within a compound statement.

no matching #if...

A preprocessor directive **#else** or **#endif** does not have a matching **#ifdef** or **#ifndef** directive.

no quote

A string constant is not terminated properly. String constants may not be split onto multiple lines.

no semicolon

A semicolon is missing from the end of a statement. This message commonly appears at the start of the following statement.

not a label

The name after the **goto** is already defined as something other than a label.

not allowed with block locals

A **goto** statement is not allowed in a function which has local variables in a block below the level of the main function block.

not an argument

A name is defined in the declarations in a function header which is not one of the arguments of the function.

not in switch

A **default** or **case** label may not appear outside of a **switch** statement.

only allowed in a loop

A **continue** statement can only be used within a loop. A **break** statement can be used in a loop or a **switch** statement.

output error

An error occurred when writing out to the output file or listing file. This normally means that the microdrive tape or floppy disk is full. This is a fatal error.

staging buffer overflow

The output buffer (used for peephole optimisation) has overflowed. This only happens on complicated statements and expressions. If you get this, simplify your code.

too many cases

There are too many cases in the switch statement for the compiler's switch table. The limit is 60 cases.

too many nested loops

There are too many nested loops - the compiler keeps track of them to handle **continue** and **while** statements. The limit is 30 loops. If you get this, simplify the code or break it up into more functions.

wrong number of arguments

The argument declarations in the function header do not match the arguments listed for the function.

APPENDIX B

Summary of Library Routines

This appendix contains a sorted list of the library functions. A summary of function parameters is included for quick reference. The section number where the function is specified is given on the right.

abort	(errcode) int errcode;	4.10.13
abs	(n) int n;	4.10.1
adate	(seconds) int seconds;	5.4.2
arc	(xl,yl,x2,y2,angle) int xl,yl,x2,y2,angle;	5.3.3
at	(line,col) int line,col;	5.2.6
atoi	(str) char *str;	4.6.1
atoib	(str,base) char *str; int base;	4.6.2
avail	(abort) int abort;	4.10.9
beep	(duration,pitch) int duration,pitch;	5.4.6
block	(width,height,x,y,col) int width,height,x,y,col;	5.2.26
border	(size,colour) int size,colour;	5.2.3
calloc	(count,size) int count,size;	4.10.7
ccargc	()	4.10.14
cfree	(pointer) char *pointer;	4.10.10
circle	(xpos,ypos,radius) int xpos,ypos,radius;	5.3.4
clearerr	(fd) int *fd;	4.3.15
cls	(part) int part;	5.2.16
csize	(width,height) int width,height;	5.2.25
curdown	()	5.2.12
curleft	()	5.2.9
curright	()	5.2.10
cursen	(switch) int switch;	5.2.5
cursor	(xpos,ypos) int xpos,ypos;	5.2.13
curup	()	5.2.11
date	(clock,datevec) int clock,datevec[];	5.4.4
delay	(ticks) int ticks;	5.4.1
delete	(name) char *name	4.3.12
dtoi	(str,num) char *str; int *num;	4.6.5
ellipse	(x,y,radius,ecc,angle) int x,y,radius,ecc,angle;	5.3.5
exec	(progname,optstr,flag) char *progname,*optstr; int flag;	5.4.12
exit	(errcode) int errcode;	4.10.13
fclose	(fd) int *fd;	4.3.3
feof	(fd) int *fd;	4.3.13
ferror	(fd) int *fd;	4.3.14
fflush	(fd) int *fd;	4.3.9
fgetc	(fd) int *fd;	4.3.4
fgets	(str,size,fd) char *str; int size,fd;	4.3.6
fill	(switch) int switch;	5.3.8
flash	(switch) int switch;	5.2.22
fopen	(name,mode) char *name,*mode;	4.3.1
fount	(fountl,fount2) char *fountl,*fount2;	5.2.17
fprintf	(fd,str,argl,arg2,...) int *fd; char *str;	4.5.2
fputc	(c,fd) char c; int *fd;	4.3.7

fputs	(str,fd) char *str; int *fd;	4.3.8
fread	(buff,size,count,fd) char *buff; int size,count,*fd;	4.10.3
free	(pointer) char *pointer;	4.10.10
freopen	(name,mode,fd) char *name,*mode; int *fd;	4.3.2
fscanf	(fd,str,argl,arg2,...) int *fd; char *str;	4.5.4
fwrite	(buff,size,count,fd) char *buff; int size,count,*fd;	4.10.4
gcursor	(xorg,yorg,right,down) int xorg,yorg,right,down;	5.3.7
getarg	(n,str,size,argc,argv) char *str;int n,size,argc,*argv;	4.10.11
getc	(fd) int *fd;	4.3.4
getchar	()	4.2.1
getpos	(fd) int *fd;	4.4.2
getwindow	(flag,pointer) int flag,pointer[];	5.2.2
ink	(colour) int colour;	5.2.21
isalnum	(c) char c;	4.8.1
isalpha	(c) char c;	4.8.2
isascii	(c) char c;	4.8.3
isatty	(fd) int *fd;	4.3.10
iscntrl	(c) char c;	4.8.4
iscons	(fd) int *fd;	4.3.11
isdigit	(c) char c;	4.8.5
isgraph	(c) char c;	4.8.6
islower	(c) char c;	4.8.7
isprint	(c) char c;	4.8.8
ispunct	(c) char c;	4.8.9
isspace	(c) char c;	4.8.10
isupper	(c) char c;	4.8.11
isxdigit	(c) char c;	4.8.12
itoa	(num,str) int num; char *str;	4.6.3
itoab	(num,str,base) int num; char *str; int base;	4.6.4
itod	(num,str,size) int num,size; char *str;	4.6.9
itoo	(num,str,size) int num,size; char *str;	4.6.10
itou	(num,str,size) int num,size; char *str;	4.6.11
itox	(num,str,size) int num,size; char *str;	4.6.12
keyrow	(row) int row;	5.4.8
left	(str) char *str	4.7.1
lexcmp	(strl,str2) char *strl,*str2;	4.7.12
lexorder	(cl,c2) char cl,c2;	4.7.13
line	(xl,yl,x2,y2) int xl,yl,x2,y2;	5.3.2
lseek	(fd,offset,from) int *fd,offset,from;	4.4.3
malloc	(count) int count;	4.10.8
nextline	()	5.2.8
otoi	(str,num) char *str; int *num;	4.6.6
over	(switch) int switch;	5.2.24
pan	(distance,part) int distance,part;	5.2.15
paper	(colour) int colour;	5.2.19
point	(x,y) int x,y;	5.3.1
poll	(pause) int pause;	4.10.12
printf	(str,argl,arg2,....) char *str;	4.5.1
putc	(c,fd) char c; int *fd;	4.3.7

putchar	(c) int c;	4.2.2
puts	(str) char *str;	4.2.3
qdosdate	()	5.4.3
random	()	5.4.9
read	(fd,buff,count) char *buff; int count,*fd;	4.10.5
readdir	(fd,fname,dirinfo) char *fname; int *fd, *dirinfo;	5.4.11
recol	(table) char *table;	5.2.18
reverse	(str) char *str;	4.7.11
rewind	(fd) int *fd;	4.4.1
rnd	(min,max) int min,max;	5.4.10
scale	(scale,xorg,yorg) int scale,xorg,yorg;	5.3.6
scanf	(str,arg1,arg2,...) char *str;	4.5.3
scroll	(distance,part) int distance,part;	5.2.14
sdate	(value) int value;	5.4.5
selwindow	(fd) int *fd;	5.2.1
sign	(n) int n;	4.10.2
strcat	(dest,sour) char *dest,*sour;	4.7.2
strchr	(str,c) char *str,c;	4.7.9
strcmp	(strl,str2) char *strl,*str2;	4.7.4
strcpy	(dest,sour) char *dest,*sour;	4.7.6
strip	(colour) int colour;	5.2.20
strlen	(str) char *str;	4.7.8
strncat	(dest,sour,n) char *dest,*source; int n;	4.7.3
strncmp	(strl,str2,n) char *strl,*str2; int n;	4.7.5
strncpy	(dest,sour,n) char *dest,*sour; int n;	4.7.7
strrchr	(str,c) char *str,c;	4.7.10
tab	(col) int col;	5.2.7
toascii	(c) char c;	4.9.1
tolower	(c) char c;	4.9.2
toupper	(c) char c;	4.9.3
trap1	(regpointer) int *regpointer;	5.1
trap2	(regpointer) int *regpointer;	5.1
trap3	(regpointer) int *regpointer;	5.1
under	(switch) int switch;	5.2.23
ungetc	(c,fd) char c; int *fd;	4.3.5
unlink	(name) char *name	4.3.12
utoi	(str,num) char *str; int *num;	4.6.7
warble	(duration,pitch1,pitch2,interval,step,wrap,fuzz,rand)	5.4.7
window	(width,height,x,y) int width,height,x,y;	5.2.4
write	(fd,buff,count) char *buff; int count,*fd;	4.10.6
xtoi	(str,num) char *str; int *num;	4.6.8

APPENDIX C

SUMMARY OF COMPILER, ASSEMBLER AND LINKER OPTIONS

Summaries of assembler and linker options are included here for easy reference, but they are not intended as a substitute for the relevant user manuals.

C.1 Compiler Options

All filenames given to the compiler in the command line (unless part of an option) are input files which are read in the order specified.

Compiler options are single letters preceded by a dash. Some of them take an parameter filename.

-M monitor: write the first line of each function to the screen as it is compiled.

-A alarm: the compiler will bleep whenever it prints an error message to the screen.

-P pause: after printing an error message to the screen, the compiler will wait for you to press the ENTER key before continuing.

-C comments: the C code is included in the output file as comments.

-D <dir> directory: the specified directory is searched for include files. Any device or directory name can be specified here. Note the parameter is mandatory in this command.

-L <name> listing: compiler listing output is sent to the named file or device.

C.2 Assembler Options

The assembler command line is in two parts. First come a number of "positional" parameters: the meaning of each parameter depends on its position within the list. There are one to three positional parameters, all filenames. These are followed by the options. An option is a word starting with a dash, and may be followed by a filename parameter.

The positional parameters are:

first	source filename
second (optional)	listing filename
third (optional)	binary filename

The options are:

-NOLIST		Do not produce a listing file.
-ERRORS	<filename>	Only send error messages and erroneous lines to the named file. This option sets the -NOSYM option.
-LIST	<filename>	Send a listing to the named file.
-NOBIN		Do not produce any relocatable binary output.
-BIN	<filename>	Send relocatable binary output to the named file.
-NOSYM		Do not produce any symbol table or cross reference.
-SYM		Produce a symbol table and cross reference.
-NOLINK		Produce absolute binary instead of relocatable binary.

The filenames specified in the options are all optional, but if specified they override the positional filenames. If a filename is not specified in either place, a default filename will be built from the source filename, with extension _LIST for listing files and _REL for relocatable binary files.

By default, the QC assembler does not produce a listing file. If a listing file is selected, a symbol table will be included by default.

C.3 Linker Options

The linker command line is in two parts, like the assembler command line. First come a number of "positional" parameters: the meaning of each parameter depends on its position within the list. There are up to four positional parameters, all filenames. These are followed by the options. An option is a word starting with a dash, and may be followed by a filename parameter.

The positional parameters are:

first	relocatable binary file name
second	linker control file name
third	listing file name
fourth	program file name

The options are:

-WITH <filename>	This identifies the control file name if no relocatable binary file is named in the positional parameters.
-NOPROG	Do not produce a program file.
-PROG <filename>	Generate a program file.
-NOLIST	Do not produce a listing file.
-LIST <filename>	Send a listing to the named file.
-NODEBUG	Do not generate a debug file.
-DEBUG <filename>	Generate a debug file.
-NOSYM	Do not produce any symbol table or cross reference.
-SYM	Produce a symbol table.
-CRF	Produce a cross reference.
-PAGELEN <number>	Specifies number of lines per page in listing.

The filenames specified in the options are all optional, but if specified they override the positional filenames. If a filename is not specified in either place, a default filename will be built from the source filename, with extension _MAP for listing files _DEBUG for debug files and _BIN for program files.

APPENDIX D

DIFFERENCES BETWEEN QC AND STANDARD C

This appendix gives a brief summary of the differences between the QC language and the de facto C standard as defined in appendix A of "The C Programming Language" by Kernighan & Ritchie.

Main Differences:

Floating point and structures and all language features relating to these types (unions, bitfields, typedef) are not implemented.

The type returned by a function in QC is always **int**, and cannot be specified otherwise.

Multi-dimensional arrays are not implemented in QC.

Other Differences: (Numbered according to K & R)

2.2 Identifiers:

External identifiers may not start with an underscore, and all 8 characters are converted to uppercase. The names of the 68000 registers are reserved.

2.4 Constants:

The digits 8 and 9 are not accepted in octal constants! The explicit long constant (letter L at the end) is not accepted.

2.5 Strings:

Strings cannot be split using \ followed by newline.

7.2 Unary operators:

Casts and the sizeof operator are not implemented.

7.13 Conditional Operator:

This is not supported in QC.

8 Declarations:

Extern and static declarations may not be used within blocks.

8.6 Initialisation:

Global pointers cannot be initialised (except *char can be initialised with a string constant).

9.7 Switch Statement:

Local variables are not allowed within a switch statement.

9.11 Goto statement:

QC does not allow goto in functions with local variables in a block within the function.

12.1 Token replacement:

Macros with parameters are not implemented in QC.

12.3 Conditional Compilation:

The #if directive is not implemented.

12.4 Line control:

The #line directive is not implemented.

QC USER MANUAL: ADDENDUM

As a direct result of GST policy for constant development and improvement of its products, QC is now supplied with the QED Screen Editor and an improved version of the COMPILE program. The QC components list shown in section 1.3 of the QC User Manual is replaced by the following paragraphs.

QC Components List

The QC Compilier is issued with the following components:

* three microdrive cartridges containing the compiler, assembler, linker, runtime library files, a screen editor and example programs

* an A5 ring binder containing the QC and QED Screen Editor User Manuals

QC1 and QC2 Cartridges

The contents of microdrive cartridges labelled QC1 and QC2 are unchanged. These are described fully in section 1.3 of the QC User Manual.

Unlabelled Cartridge

The contents of the microdrive cartridge unlabelled is as follows:

QED	the QED Screen Editor for preparation and editing of QC (or assembler) programs
COMPILE_210	an improved version of the COMPILE program that invokes the QED editor, displays directory names and (if a single floppy is used) a free/used sector count and file sizes in bytes
COMPILE_210_C	the source code (in QC) for the improved COMPILE program

Use of the unlabelled Cartridge

The cartridge should be alternated with QC1 in MDV1_ using QC2 in MDV2_ to hold the source file(s) that are alternately edited and then compiled. If you have floppy disks, you are strongly recommended to copy all the software onto disk in FLP1_ (which will also have sufficient room for your source files). Remember to back up your cartridges prior to use.