C68
C Compiler for the Sinclair QL

INTRODUCTION

The C68 Compilation System provides a Public Domain C compiler for use under the QDOS operating system. It is a full C implementation that includes all items mentioned in the "Kernighan and Richie" C definition.

There is full support for all common data types such as int, char, short, long, float and double as well as more esoteric types such as "typedef" and "enum". Structures and unions are supported for those who want more complex data types.

The C68 Compilation System includes everything that is needed to produce a running C program, including a simple source code editor for those who do not already have one. However editors is an area where every user seems to have their own personal preference, so you are at prefect liberty to use an alternative Many C programmers are likely to already have a suitable editor that they like and love!

THE COMPILATION PROCESS

The C68 Compilation System is implemented in the style that is common on Unix systems where the compilation process is broken into a number of discrete phases. These are:

CPP	pre-processor
C68	compiler
AS68	assembler
LD	linker

The user does not normally run these programs directly. Instead they are front-ended by the CC command. This will examine the parameters it is provided with, and will run the appropriate underlying programs.

The job of the CPP pre-processor is to take the C source provided and scan it executing all the C directives (the ones that start with the # symbol) such as #include and #define statements. This produces C code with all these directives removed that is suitable for input to the main compilation phase.

The output from the CPP pre-processor is then input into the C68 compiler. At this stage all the syntax analysis of the user's program is done, and code generated. The C68 compiler outputs assembler source code.

The assembler source code is then converted to SROFF (Sinclair Relocatable Object File Format) by the AS68 assembler.

Finally the LD linker is used to combine the user's program module(s) with standard library modules that are supplied as part of a C implementation.

It is tedious to have to keep typing in all the parameters required to compile a particular program (particularly if it consists of multiple modules). The C68 Compilation System provides the MAKE command to allow this process to be automated.

THE LIBRARIES

A key part of any C implementation is the libraries that are supplied with it. The more extensive the libraries, the easier it is for the programmer to implement any particular facility.

One of the strengths of the C language is the ease with which programs can be ported betwen different computers and operating systems. This is only true, however, if both systems have comparable (and preferably compatible) library routines.

The standard C library supplied as part of the C68 Compilation System includes all routines defined by Kernighan and Richie; all routines defined by the ANSI standard; most of the routines commonly implemented by the LATTICE C family of compilers; and a large number of library routines commonly encountered in the Unix environment.

For those who want to access QDOS, access is provided to all of the QDOS operating system calls. There are standard routines to satisfy many tasks commonly encountered by programmers (e.g. a routine to obtain a sorted directory listing, or a list of files matching a wildcard pattern).

Additional libraries cover more specialist areas such as MATHS routines and debugging aids. Libraries are under development to cover areas such as QRAM support and Semaphore handling.

SOURCE

All elements of the C68 compilation system are in the Public Domain. For those who are interested the full source of all components is available.

Except for some of the library routines, the rest of the C68 Compilation system is itself written in C. The C68 Compilation system is in fact used to compile itself!

HARDWARE REQUIREMENTS

The one drawback of the C68 Compilation System is that it will not run on an unexpanded QL. The minimum requirements are 256Kb of memory and at least one 720Kb floppy disk drive. Additional memory and/or disk drives are highly desirable.

USING THE C68_MENU FRONT-END

The simplest way to use C68 is via the C68_MENU program supplied in the C68 System disk. Using this program is largely intuitive, but full details are contained in the file C68MENU_DOC on the documentation disk.

A suitable BOOT file is supplied on the "RUNTIME 2" disk. You may if you wish, simply boot your system with this disk in FLP1_, and this BOOT file will be used. Alternatively, you may wish to make your own tailored BOOT file, using this one as a model.

It is only necessary to read the remainder of this document if you intend to run C68 from the SuperBasic command line. Having said that, even if you are going to use C68_MENU, it is still a good idea to at least look through it to get an idea of what is happening behind the scenes.

PREPARING TO USE C68

The C68 will use Toolkit 2 default directories. This is very convenient as it allows you omit the device and directory part of any filename.

Therefore, the first thing is to set the default directories. The recommended settings are as follows:

```
    DATA_USE FLP2_
    PROG_USE FLP1_
```

You can check the current settings of the DATA_USE and PROG_USE directories at any time by typing in the command:

```
    DLIST
```

You can now run with the C68 System Disk (RUNTIME 1) in FLP1_, and your work disk in FLP2_. The defaults built into C68 will now be looking for files in the correct place.

HARD DISK USERS

The instructions outlined in this document assume that you have a twin floppy disk system. Hard disk users follow the same principles, but set the DATA_USE and PROG_USE directories to point to the appropriate hard disk directories.

COMPILING PROGRAMS

You can now compile any program by simply typing in the command of the form

```
EX CC;"-v -oPROGRAM PROGRAM_C -lm"
```

All parameters except the source file name can actually often be omitted. The -o option is used to name the file to contain the final program where "PROGRAM" is the name of the program. If you omit the -o option then your final program will be called A_OUT. Do not let your program name finish with _I, _S or _O as these have special meaning within C68. The -v parameter makes CC display the command line that it is using to run the various compiler phases that it is running for you. Finally, the -lm parameter is used to cause the linker to search the LIBM_A library in addition to the standard C library. This is needed if you want to print floating point numbers as the versions of the print routines that support this are held in the maths library. If you have omitted the -lm option and DO try to print floating point, you will get a message saying "no floating point" displayed instead of the number you expected.

RUNNING THE COMPILED PROGRAMS

You can run now run the generated program. You start the generated program by a command of the form

```
    EXEC_W flp2_PROGRAM
```

If you omitted the -oPROGRAM parameter to CC this would be

```
    EXEC_W flp2_A_OUT
```

Note that the final program will have been put into the DATA_USE directory, so it is not possible to default the directory part of the filename.

WHAT NEXT

By this time you are should be able to compile and run simple programs. The next stage is to examine the C68 documentation in more detail. The README_DOC file will give you a good idea of the contents of each file, so you can decide which ones you want to read first. Some of the most important files to read early will be the OVERVIEW_DOC and the INTRO_DOC files.

Eventually you are likely to want to print out most of the documentation for reference purposes. There are, however, several hundred pages so this is a non-trivial task, although I think that you will find it to be well worth the effort.

Many of the documents now include a change history of when the last significant change was made. This can help uses who are upgrading releass to decide if a document has changed significantly. The footer also includes the date of the last change even if it was only a trivial one.

It is intended that starting with the 4.20 release, the CHANGES_DOC file will contain a list of what documents have had significant changes since the previous release.

C68 With QDOS

This document describes how the C68 implementation of C under QDOS has been adapted to use the features of the QDOS and SMS operating systems. It also describes what has been done to keep maximum compatibility with C programs that have been written to run under Unix - the home of the C language, and still the commonest source of freely available C source code.

The "C68 for QDOS/SMS" system has been supplied so that its default mode of operation is to function as far as possible in a manner similar to a standard C program running under Unix or MSDOS. However, there are a number of standard options (which are discussed later in this document) for changing the default behaviour and adapting it to your specific requirements.

PROGRAM NAME

When it is started up, the a C68 program copies its name into the first part of the program space. This is so that QDOS/SMS commands for listing the jobs running can display a sensible name for the job. To set the name of your program declare it by including the line

```
char _prog_name[] = "program_name";
```

in one of your source files outside any function declarations. Note that this is NOT the same as

```
char *_prog_name="program_name";  /* This is an ERROR */
```

as the above declares a pointer to a character array, and the code that copies the program name assumes that _prog_name is the base address of a character array (not the same thing!).

If no program name is given a default name of C_PROG is used, so _prog_name need not be defined if you don't mind your program being called C_PROG.

ARGUMENT HANDLING

The command line is parsed into separate elements so that it may be accessed via the argv[] array. By UNIX convention, argv[0] always points to the name of the program (this is set to point at _prog_name). The other arguments (if any) are taken from the command line and are put into the argv[] argument array. Each argument in the command line should be separated from the others by one or more whitespace characters.

If you want to include space characters within an argument this can be done by surrounding the argument value with either single or double quotes. Therefore to get an argument array of :

```
argv[0] = C_PROG
argv[1] = test
argv[2] = of multiple
argv[3] = arguments
```

then you would invoke your program as follows:

```
    EX MY_PROG;'test "of multiple" arguments'
```

Note that the quotes surrounding the words are NOT copied into the argument string. If you want to include otherwise forbidden characters in an argument such as ', or ", or any of the special argument characters =, %, >, <, (covered later) then they may be included by prefixing them with a \ character. A \ character may itself be included by using \\. Therefore the program invoked by :

```
    EX MY_PROG;' wombat \"quote "have big" \\ears'
```

would have an argument array of

```
    argv[0] = C_PROG
    argv[1] = wombat
    argv[2] = "quote
    argv[3] = have big
    argv[4] = \ears
```

Arguments that start with the special sequences <, >, >&, >>, >>&, = and % (which are used for special purposes as identified later in this document) are not copied into the argument array as they are stripped from the command line when they are acted on. This will happen automatically before control is passed to the user code.

You can also use standard C escape sequences within the command line. These can be character escape sequences (such as \t for a tab character), octal escape sequences (such as \009) or hexadecimal escape sequences (such as \x09). Use these escape sequences if you want to put one of the special reserved characters mentioned in the previous paragraph into a command line parameter.

If you know that your program does not accept any parameters (although it may still accept the sequences identified by the special characters), then you can include a line of the form

```
    void (*_cmdparams)() = NULL;
```

in your program outside any function declarations. This will stop the code that is used for parsing the parameters in the command line from being added to your program, and will reduce its size accordingly.

REDIRECTION

The C68 system allows a UNIX compatible style of redirection symbols to be used to redirect the programs input and output streams away from the normal console channel. Their action is as follows:

`< filename` This redirects the standard input (file descriptor 0 or 'stdin') of the C program so that all reads to it are read from the designated file (or device). If this is not present in the command line then the standard input defaults to CON_.

`> filename` or `1>filename`	This redirects the standard output channel (file descriptor 1 or 'stdout') only, so that it writes to the designated file or device name. If the file doesn't exist, it creates it, if the file does exist it is truncated. If this option is not given then the standard output defaults to the same CON_ channel as stdin or, if this has been redirected, to a new CON_ channel.
`>> filename` or `1>>filename`	This redirects the standard output channel to the given file or device name. If the file does not exist it is created, if it does exist it is opened for appending. All writes will be done to the end of the file, so no existing data will be overwritten.
`>& filename`	This redirects both the standard output and standard error (file descriptor 2 or 'stderr') channels to the designated file or device. The file is created if it doesn't exist, or truncated if it does.
`>>& filename`	This redirects stdout and stderr to filename, creating it if it dooesn't exist, but opening it for appending if it does.
`2> filename` or `& filename`	These redirect the standard error channel (file descriptor 2 or 'stderr') only, so that it writes to the designated file or device name. If the file doesn't exist, it creates it, if the file does exist it is truncated. If this option is not given then the standard output defaults to the same CON_ channel as stdout or stdin, or if this has been redirected, to a new CON_ channel.
`2>> filename` or `&> filename`	These redirect the standard error channel to the given file or device name. If the file does not exist it is created, if it does exist it is opened for appending. All writes will be done to the end of the file, so no existing data will be overwritten.

If you know that your program will not be have its channels redirected from the command line (or wish to inhibit this capability), then you can include a line in your program of the form

```
     long (*_cmdchannels)() = NULL;
```

outside any function declarations. This will stop the code that handles this capability from being included in your program and reduce its size accordingly.

PASSING CHANNELS FROM SUPERBASIC

C68 allows channels to be passed from SuperBasic to a C68 program via the command line. This is done by preceding the argument list with the channels to be passed.

```
Eg.  EXEC c_prog,#1,#2,#0;"parameter list"
```

The first channel is allocated to stdin, and the last one to stdout. If three channels are supplied then the second one is allocated to stderr. These channels are available at all I/O levels (see later). Additional channels are initially available only at Level 1 I/O (described later). They are allocated file descriptors starting at 3.

If you know that your program will not be passed channels from SuperBasic (or wish to inhibit this capability), then you can include a line in your program of the form

```
     long (*_stackchannels)() = NULL;
```

outside any function declarations. This will stop the code that handles this capability from being included in your program and reduce its size accordingly.

CONSOLE SIZE AND PLACING

Normally a C68 program will need to open a Console channel to be used for stdin, stdout, and stderr (assuming none of these have been re-directed). This Console channel is opened using the name defined in the global variable _conname. The default is equivalent to defining

```
          char _conname[] = "con";
```

If the default is not satisfactory, then alternative details can be provided by defining this global variable in your own program with a suitable entry outside any function defintion.

NOTE The settings for the size and placing of the console window used for stdin and stdout will normally be overridden by a console initialisation routine as mentioned below. If you have disabled the console initialisation routine, then the settings from the initial open of the console will remain in force.

CONSOLE INITIALISATION

When a C68 program starts up and it has not had its standard output redirected, then it will have a console channel set up for the stdout device as mentioned above. Various options are then available for initialisation of the console window.

The default initialisation that is performed if no alternative is exlicitly specified involves setting up a window according to the values in the global structure _condetails. This is data item of type WINDOWDEF_t (as defined in the sys/qdos_h header file). For the default set of values, see the definition of the global variables at the end of the LIBC68_DOC document.

Alternatively there is a library routine available which does more sophisticated initialisation. As well as the initialisation that is performed by the default routine mentioned above, it will in addition add a title bar across the top of the window that gives the program name. The active size of the window will be reduced accordingly. This routine is invoked by including the following lines in your program at the global level (i.e. outside any function definition):

```
void consetup_title(chanid_t, struct WINDOWDEF *);
/* above line not needed if qdos.h or sms.h included */
void (*_consetup)() = consetup_title;
```

If you are running under the Pointer Environment then this library function will also define an 'outline window' that will be the same size as this console window (including borders and title bar). This will cause the Pointer Environment to tidily save and restore the screen as you switch between jobs.

If your program is designed so that the Pointer Environment is mandatory, then there is a more sophisticated module available in the LIBQPTR_A library called consetup_qpac.

If you wish no automatic console initialisation to be done (perhaps because you wish to do all of this in your own program code) then you should include the following line in your program at the global level (i.e. outside any function definitions):

```
void (*consetup)() = NULL;
```

This will have the effect of disabling any automatic console initialisation from taking place.

If none of the above options suit your requirements, then you can provide an alternative routine to be used in place of one of the standard ones. To provide an alternative routine you proceed as in the example above for using the consetup_title() routine, but substitute the name of your routine for 'consetup_title'. The console initialisation routine should have should have a prototype of the form:

```
void  my_console_routine (chanid_t console_channel);
```

PAUSING WHEN A PROGRAM TERMINATES

If you are running under a multi-tasking environment (such as the QJUMP Pointer Environment) then it is convenient if the program pauses to give you a chance to read any messages before it exits. This is the default behaviour for C68 that is built into the library routines. The message that is displayed is equivalent to defining the global variable

```
char  *_endmsg = "Press a key to exit";
```

You can change the message by defining the above global variable in your own program with a different text. If you do NOT want the program to halt with a message on exit, then you must set the _endmsg global variable to NULL at any stage before you exit your program.

The termination message will also be suppressed if stdout has either been redirected, or if it has been passed as an open channel from another program. This stops a chain of programs sharing the same output channel each outputting their own termination message.

By default when the termination message is displayed, the system will wait indefinitely for the user to press a key. The wait duration is actually defined by the setting of the _endtimeout global variable. The default value supplied is equivalent to specifying

```
    timeout_t _endtimeout = -1;        /* Wait forever */
```

in your program. A positive value means wait that number of 1/50 seconds.

MEMORY ALLOCATION

When a C program is loaded into memory and starts, it first relocates itself to run at the load address. It then calls QDOS to allocate a memory area that is used as the base of the programs own private heap and stack areas.

The heap area is used to satisfy any dynamic memory allocations made via the malloc() library call. It is also used by many of the library routines if they need additional workspace.

The stack area is used when a program function is called. It holds any parameters passed, and the return address for when a function completes. It also holds all local variables defined within program functions.

The heap can be expanded by allocating new areas from QDOS, but the stack must never outgrow its initial allocation. Outgrowing the stack area can have dire consequences, sometimes even causing system crashes. There are checks when memory is allocated from the heap to check that the stack pointer has not exceeded the area allocated. However there is no check made when the stack expands as a result of calls to program functions. In practise the default stack is sufficient for all except the most demanding programs.

The programmer can set the default values that will be used for the sizes of memory areas. If the programmer declines to provide any values then defaults will be set that are suitable for the vast majority of programs. In addition, the user can use run-time values to increase the sizes of the areas.

The programmer sets up these values by defining them as variables outside any function declaration (to ensure the variables are global in scope). The default values are equivalent to the programmer defining:

```
    long _stack = 4L*1024L;       /* size of stack */
    long _mneed = 4L*1024L;       /* minimum requirement */
    long _memmax = 9999L * 1024L; /* maximum allowed */
    long _memincr = 4L * 1024L;   /* increment size */
    long _memqdos = 20L * 1024L;  /* minimum for QDOS */
```

These values of these variables are used by the C68 system in the following way:

`_stack`	This is the amount of area to be allocated as the program stack. The program must never exceed this value while it is running, or undefined effects are likely to can occur.
`_mneed`	This is the amount of heap space that is initially allocated to the programs private heap.
`_memmax`	This is the maximum amount of heap memory that a program is ever allowed to grab. The default value is so high that this limit effectively does not apply.
`_memincr`	To avoid excessive heap fragmentation C68 programs manage their own private heap, and only allocate themselves more memory from QDOS when the private heap is exhausted. This is the allocation size in which new memory is requested from QDOS.
`_memfree`	This is an alternative way of controlling the maximum memory allocated. The program is never allowed to allocate itself more memory if it would reduce the free memory left in the machine below this value.

As was mentioned earlier, in the vast majority of cases you can merely accept the default values.

Situations sometimes arise, however, in which the user wishes to increase some of these values at run time. This can be done by including one or both of the command line arguments:

```
     =ssss %hhhh
```

where ssss is a decimal number denoting the amount of stack given to the program (equivalent to setting the _stack global variable), and hhhh is a decimal number denoting the amount of heap (equivalent to setting the _heap global variable).

In the case of the %hhhh option the _memmax value is also set to this value so that the program will not be allowed to increase its memory allocation any further. Four digits are used above for illustration purposes only; in reality, the only limit is the amount of memory in the machine).

DISABLING SETUP OF STANDARD C ENVIRONMENT

You may have noticed that even if you write a very simple program that is only a few lines long that it will still be around 12-14Kb in size. This is not because C is inefficient, but because the default startup code that is included in a C program includes a lot of library routines. These are used to set up a standard environment for the C programmer. This includes doing all the following:

- Processor identification and program relocation
- Parsing the command line parameters
- Initialising the standard default channels to be used for input and output.
- Setting up Environment variables
- Setting up Dynamic memory support

There may be times when you do not need all this done for you. This would typically be the case if your program was only going to make use of direct QDOS/SMS calls for input/output and memory allocation. In this case you can specify that you do NOT want the default C environment set up and reduce your program size by about 12Kb. You do this by including the following line within your program at global scope:

```
(*_Cinit)() = main;
```

In this case the only startup code that will be executed is that involved with doing the Processor type identifiaction and the program relocation. Control will then be passed direct to your main() module. The following parameter values will be passed to main:

`argc`	This will always be set to 1
`argv[0]`	This will always point to the program name
`argv[1]`	This will point to the program stack that was in use at initial program start-up. This will allow you to access any information that was passed to the program on the stack.

AUTOMATIC HANDLING OF FOREIGN FILENAMES

Many systems make use of special characters in filenames to act as directory seperators and to identify the 'extension' part of a filename (which traditionally identifies the file type). As an example Unix uses the '/' (slash) character as a directory seperator and the '.' (full stop) character to identify the start of the file extension. MSDOS uses '\' (back slash) as the directory seperator and also uses '.' as the file extension character.

QDOS and SMS traditionally use the '_' (underscore) character for both of these purposes.

If you are porting a program that is meant to run on a foreign system (such as Unix or MSDOS) it can be a real nuisance to find everywhere where a filename is manipulated to change it to the QDOS/SMS standard. This problem can be obviated by including the following lines in your program at global scope:

```
#include <fcntl.h>
int (*_Open)(const char *, int, ...) = qopen;
```

This will then automatically convert any attempts to open files using the foreign filename standard into calls that conform to the QDOS/SMS standard. If you want a more detailed description of exactly what happens refer to the description of the qopen() library routine, the _Open vector and the _Qopen_in and Qopen_out global variables which are all described in the LIBC68_DOC file.

INPUT/OUTPUT LEVELS

The input/output facilities under C68 are structured as a series of Levels depending on the level of abstraction that is wanted. These levels can be summarised as:

Level 2	Generic C input/output
Level 1	UNIX compatible input/output
Level 0	QDOS specific input/output

Each level has library calls that can open, read and write files. Opening a file at a specific level also does implicit opens at the lower levels, but not vice versa. This will probably be clearer when you have read the following descriptions of each level.

C LEVEL FILE POINTERS (Level 2 I/O)

In standard C, programs communicate to the outside world via File Pointers. This interface is completely supported by C68. It is the level that you should work at if you are new to C, or if you want to write programs that will be portable to other systems. This level of interface is referred to in the C68 documentation as Level 2 I/O.

A point to note is that C does its own internal buffering, and the buffers are only flushed when end-of-line is reached. For console and screen channels this is often not convenient. You can disable the buffering by using a statement of the form

```
        setbuf (file_pointer, NULL);
or      setnbf (file_pointer);
```

(the second option is less portable as it is not supported by all systems) in your program before you do any reads or writes to the file. You will find that you need to do this also on any file in which you are going to mix the Level 2 I/O with either Levels 1 or 0 as defined in following sections.

An alternative solution is to ensure that you have always done a fflush() operation on the file in question before you do level 1 or level 0 I/O. The advantage of this is that the C level I/O is more effecient, against the fact that that you have to remeber to do the fflush() calls to force output to be displayed before using any level 1 or level 0 calls.

If files are opened at Level 2, then Levels 1 and 0 are also implicitly opened.

UNIX LEVEL FILE DESCRIPTORS (Level 1 I/O)

This next section is only relevant if you are trying to use the C68 I/O interface that corresponds to the Unix I/O interface. Unix systems have an I/O interface available to C programs which is lower level than the standard C interface, and maps directly onto underlying operating system calls. C68 provides library routines which mimic the Unix system call interface. This helps with porting programs from Unix systems to QDOS with C68. This level of interface is refered to in C68 as Level 1 I/O.

Under the Level 1 I/O interface C communicates to the outside world via file descriptors. These are positive integers starting at zero that specify output channels. By convention the stdin, stdout and stderr correspond to the file descriptors with values 0, 1 and 2 respectively. You can always obtain the Level 1 file descriptor associated with a file opened using Level 2 I/O by using the library call

```
level_1_fd = fileno (level_2_file_pointer);
```

Note that it is not possible to go the other way and obtain the Level 2 File Pointer from the Level 1 File Descriptor. Always therefore use the open type appropriate to the highest level of I/O you wish to perform on a file.

The level 1 file descriptor is different from the 32 bit QDOS channel id. The QDOS channel id can be obtained from the Level 1 file descriptor by using the library call:

```
qdos_id = getchid(level_1_fd);
```

Given this information, how does this version of C handle the QDOS window interface? Well, as much as possible it ignores it and tries to run as a "glass teletype", which is what most UNIX programs expect. However, there are some useful pieces of information about the way screen I/O is handled.

If a file descriptor (hereafter known as an 'fd') is opened onto a CON device by the open() call, it is by default opened in 'cooked' mode. That is; all reads will wait until the required number of characters are available (or enter is pressed); all characters typed are echoed on the screen; full QDOS line editing is available; all pending newlines are flushed after the read call completes.

If the mode is changed to 'RAW' (fd opened with O_RAW flag set, or fcntl() or iomode() library calls done on a fd channel) then all reads are done without echoing on the screen; no line editing is performed; no pending newlines are flushed; no cursor is enabled; and the results of a one byte read are available immediately. So to read one character immediately with no echo and receiving all cursor and function key presses (a perennial problem for C programmers writing interactive software), just open the fd in O_RAW mode (or change an already open one to RAW), then use

```
read( fd, &ch, 1);
```

to read a character.   Note that under C68 characters with an internal value higher than 127 will be returned as a negative value as char is a signed value.

Normally all I/O is done with an infinite timeout, but if you are running in supervisor mode or just want reads and writes to return immediately you can force the level 1 I/O calls (those that use fd's) to use a zero timeout by either opening the file descriptor with the O_NDELAY flag set, or doing a fcntl() library call to set the O_NDELAY. This forces calls to return immediately if they are 'not complete' with the appropriate error.

As the QL uses newline ('\n' ascii 10) characters to designate End-Of-Line (as do UNIX systems) then there is no option to open a level 2 file (pointed to by a FILE pointer defined in stdio.h) in 'translate' or 'binary' mode. Such calls, eg. fopen("file", "rb"); will succeed but the binary flag will be ignored. By default, level 1 files may be opened in O_RAW mode by setting the _iomode external variable to O_RAW before any files are opened.

To change an open file descriptor the fcntl() or iomode() calls may be used (defined in fcntl.h). The fcntl() call changes the flags set in the underlying control structures according to the values of the flags defined in fcntl.h. Eg. to set a channel in raw mode read the value of the _UFB flags using fcntl() then set O_RAW mode with the same call.

```
    flags = fcntl( fd, F_GETFL, 0);
    fcntl( fd, F_SETFL, flags | O_RAW );
```

The iomode() call has a similar effect to the above but toggles the state of any flag. Eg. if a fd is set to O_RAW, doing

```
    iomode( fd, O_RAW);
```

will set it to raw mode, then doing the same iomode() call on it again will set it back to cooked mode.

QDOS CHANNEL IDENTIFIERS (Level 0 I/O)

The C68 system allows you to call the underlying QDOS I/O system directly by-passing the higher levels of I/O. This level of interface is referred to as Level 0 I/O.

To access QDOS directly you need the QDOS channel id. You can obtain the QDOS channel id associated with files opened using Level 2 or level 1 I/O by using the library routines:

```
    qdos_id = fgetchid (level_2_file_pointer);
    qdos_id = getchid (level_1_file_descriptor);
```

If you initially open a file at level 0, it is also possible to create a level 1 file descriptor or a level 2 file pointer to allow you to manipulate the file at the higher levels. To do this you can use the library routines:

```
    level_2_file_pointer = fusechid (qdos_channel);
    level_1_file_descriptor = usechid (qdos_channel);
```

Note that the fusechid() call will also create a corresponding level 1 file descriptor (which you can obtain, if required, by using the fileno() library call).

It is important if you mix QDOS level I/O with one of the higher levels that you have disabled any buffering at the higher level. Failure to do this is almost certain to result in output appearing in an unexpected order. For convenience, the stdout file is unbuffered by default.

INPUT TRANSLATION

Many Unix systems use have special keyboard sequences to input control characters from the console. The normal QDOS console driver does not recognise such control sequences. Therefore when input is being done from a console device a special routine (called _conread()) has been provided that is called at the appropriate point. The special characters that are acted on are:

CTRL-D	Passes an EOF code to the user program.

N.B Programs which use the LIBCURSES or the LIBVT libraries get alternative versions of the _conread() routine. Refer to the documentation on these libraries for more details.

If the user knows that this input translation will not be required, then the program size can be reduced by including the following code in your program:

```
int (*_conread)() = NULL;
```

If user programs wish to provide alternative input handling to the default, then they can provide there own version of the _conread() routine. The user supplied routine would be included by using a line of the form:

```
int (*_conread)() = my_read;
```

The code for the existing _conread() routine (which is provided on the C68 SOURCE 1 disk) should be used as a guide on how to go about this.

OUTPUT TRANSLATION

The C language defines a number of escape sequences that can be included in the user program. These range from the common ones such as \n for newline to more esoteric ones such as \t (tab), \b (backspace) and \a (alarm). The normal QDOS console driver does not recognise such escape sequences. Therefore when output is being done to a console device a special routine (called _conwrite()) has been provided that is called at the appropriate point.

N.B Programs which use the LIBCURSES or the LIBVT libraries get alternative versions of the _conwrite() routine. Refer to the documentation for these libraries for more details.

If the user knows that this output translation will not be required, then the program size can be reduced by including the following code in your program:

```
int (*_conwrite)() = NULL;
```

If user programs wish to provide alternative output handling to the default, then they can provide there own version of the _conwrite() routine. The user supplied routine would be included by using a line of the form:

```
        int (*_conwrite)()=my_write;
```

The code for the existing _conwrite() routine (which is provided on the C68 SOURCE 1 disk) should be used as a guide on how to go about this.

DIRECT CONTROL OF KEYBOARD INPUT

It is possible to get at the keyboard input before it is seen by any of the standard C library code. A typical use of this vector might be if you want to intercept particular keystrokes and act on them independently of your main C program. To do this you include a line in your program at global scope of the form:

```
        int  (*_readkbd)(chanid_t,timeout_t,char *) = myroutine;
```

The default setting of this vector is to point to the operating system call (io_fbyte() for QDOS and iof_fbyt() for SMS) that gets a single byte from the keyboard.

An example of how this vector can be used is shown in the readmove() routine described in the QPTR part of the standard C library.

IMPLEMENTATION OF PIPES

Owing to the closeness between QDOS and UNIX the concept of opening pipes between processes is easily accommodated. The only problem is that pipes have to have both ends opened at the same time (there is no concept of 'named' pipes under QDOS) which means that both ends of a pipe are owned by the job that opened them (usually the parent job in a tree of jobs).

This means that after opening the pipes the parent job must stick around until all use of the pipes by the child jobs have finished, or the child jobs get a rude shock when they try to read or write to a closed pipe after the parent has terminated (assuming the child jobs are made independent of the parent, i.e. owned by SuperBasic).

Otherwise the pipe() call acts as normal, creating an input and output pipe connected to each other. The size of the output pipe is specified in the global variable _pipesize, which is normally set to 4096 bytes, but can be changed by the program by including the line:

```
        int _pipesize = required_size; /*Outside any function */
```

RUNNING CHILD JOBS

The standard libc_a library has calls that mimic the UNIX fork() and exec() calls closely, but not exactly. The deviations are due to differences in the underlying operating system.

The exec() calls load and activate a job and the parent job that called exec() waits until the child job has finished, and reports its error code. This is unlike the UNIX exec(), which overlays the currently running program with another.

The fork() calls load and run a child job whilst the parent program continues to run, returning the QDOS job id of the child job. This is in contrast to the UNIX fork() which duplicates the running process.

After a fork() call the parent job may choose to wait for any of its child jobs to terminate (an example of wanting to do this would be if after creating pipes between two child jobs; the parent needs to wait for both jobs to finish before closing the pipe that it owns. The wait() call allows this. It returns -1 immediately if there are no child jobs, otherwise it waits for one of its child jobs to finish (it actually puts itself to sleep waiting for a child termination - it does not busy-wait) and then it returns the error code from the newly terminated job, and the job id of the newly terminated job. An example of its use is:

```
/* Start a load of child jobs */
     ...
     fork( .. );
     fork( .. );
     fork( .. );
     ...
while( wait( NULL ) != -1)
           ; /* Wait for all jobs to finish */
```

All jobs started by fork() or exec() are started with a priority of _def_priority. This is a global variable in qlib_l, and so may be changed from its starting value of 32. Channels may be passed to jobs in the fork and exec calls, these are used as the standard in, out, error and further channels. Note that these are passed according to toolkit 2 protocols. These state that the first channel passed is the job's standard input, the last channel passed the job's standard output, and all others are available to the job from file descriptor 2 (stderr) and up. This means that to pass fd 2 as stdin, fd 4 as stdout, fd 1 as stderr, and also pass channels 0 and 3, the channels array passed to fork or exec should be:

```
chan[0] = 5; /* Number of channels to pass */
chan[1] = 2; /* stdin */
chan[2] = 1; /* stderr */
chan[3] = 0;
chan[4] = 3; /* General channels */
chan[5] = 4; /* Stdout */
```

Note that the actual QDOS channels that the fd's use are passed on the stack. This means that for fork() calls, where the parent and child jobs are both active at the same time, then any changing of the channels position by read(), write() or lseek() calls will alter the read/write pointer on the channel for BOTH jobs. Thus it is better not to access channels given to child jobs whilst the child jobs are active, unless you are very careful about the consequences. One way of doing this is to close the channels that a parent has just passed to a child, to prevent the parent accessing them again.

DEFAULT DIRECTORIES FOR OPENING FILES

The I/O routines in the C68 libraries will all make use of Toolkit 2 default directories when attempting to open files. The default directories will be used any time a full absolute path name is not given for a file.

The DATA_USE directory is used for normal open statements within programs, and the PROG_USE directory for attempts to access system files (such as child programs). Extended forms of the file opening functions (fopene() and opene()) are also available to allow the programmer to specify exactly which default should be used in any particular case.

If a program is started from SuperBasic, then it inherits its default directories from SuperBasic. The program can subsequently issue calls to change them (using the chdir(), chpdir() and chddir() library functions). These will affect subsequent file open calls made by the program, but will leave the SuperBasic settings unchanged.

If a C68 program starts another child C68 program, then the child program inherits the current default directory settings of its parent job, rather than the current settings at the SuperBasic level.

WARNING. There can be a problem if you try and use a filename that could be confused with a device name. The C68 system will probably try and open the device rather than the filename that you expected to be used. This is because c68 will try and open the device first.

This means that names that begin with items like the following (probably followed by and underscore and further text) are likely to not work correctly:

pipe
ser
spell (if you use the QJUMP Spell checker)

The same would apply to any other name that could be confused with a simple device on the system in question.

ENVIRONMENT VARIABLES

The C68 system allows for the use of Unix style Environment variables. For more detail on how environment variables are used look at "Environment Variables" document (in the file ENVIRON_DOC).

If a program is started from SuperBasic, it inherits its environment variables from SuperBasic. The program can subsequently issue calls to change its own environment variables (using the putenv() function) without affecting the settings at the SuperBasic level.

If a C68 program is a child of another C68 program, then the child program inherits the current environment variables of its parent job, rather than the current superBasic settings.

C68 follows UNIX convention in that the main() function in the user program is passed a third parameter (char *argp[]) which gives a user program details of its initial environment. The argp parameter is an array of pointers terminated by a NULL pointer. Each entry in this array points to a single environment variable. These environment variables are each C strings of the form NAME=value.

WARNING. The argp array will no longer be valid if the user program performs any putenv() library calls. The global environ variable can, however, be used instead as this is updated by putenv() calls.

READING QDOS DIRECTORIES

There are various sets of routines for reading directories. The simplest are the Unix compatible set. These will automatically reformat the information to the Unix style for directory entries.

There are two other sets of library routines that deal with reading QDOS directories. One set of these (the ones that return struct DIR_LIST pointers) are for producing sorted lists of filenames, sorted on any criterion (as QRAM produces).

The other set are for scanning a directory file by file (the read_dir type of calls). These take less memory but whilst the directory is open no new files can be created by any job, as the directory structure is locked by the job owning the channel id to the directory. Note that this means the calling job itself as well, so opening a directory file, then trying to create a new file in that directory will cause the job to deadlock forever (the create file call is waiting for the directory to be released, which cannot happen until the create file call finishes!).

These directory reading calls take wildcard parameters (as described later in this document) and will return short names if the current data directory is being read (eg. if the current data directory is flp1_test_, containing files flp1_test_file1, and flp1_test_file2, then reading the current directory will return the names file1 and file2, rather than the complete name).

Also directory searches may be limited on file attributes (program, data, directory etc.) Defined constants are provided in qdos.h to allow any range of file attributes to be selected on a directory read (OR'ing the required types together allows more than one type to be recognised by the reading routines).

QDOS AND SMS TRAPS

The full range of QDOS and SMS trap calls are provided as separate routines in the C68 libraries. Several of the most useful vectored routines are also available although not all of them.

The graphics calls have been expanded to work with both integer and C double precision floating point arguments. As a rule, the sd_i type graphic routines take integer arguments, whereas the sd_ type routines take double arguments.

Conversion routines have also been provided. There are some that convert short and long integers and double precision floating points to and from QDOS/SMS floating point. There are another set that convert between C and QL string formats.

For any extra routines that are needed to call toolkits etc. the routines qdos1(), qdos2(), and qdos3() are provided that allow direct access to the QDOS traps.

The QDOS/SMS system variables are pointed to by an external variable _sys_vars. This is set at startup time to point to the base of the system variables.

There may occasionally be times when you need to go into supervisor mode (thus disabling multitasking). To do this the _super() and _user() calls are provided. The _super() call sets your program into supervisor mode, and the _user() function exits from supervisor mode.

Note that, as supervisor mode uses a different stack to user mode it is VITAL that your program does not return from the function that called _super() before calling _user() first. _user() puts the program back into user mode and restores the program's ordinary stack. The _super call is not re-entrant, ie. If you call it when you are in supervisor mode, your program will crash (although we hope to lift this restriction in a future release of C68).

It is bad practice to use Supervisor mode unnecessarily. Therefore you should avoid using supervisor mode unless you really need it.

USING FLOATING POINT

The versions of printf and scanf (and the variants of these) that support floating point are considerably larger than those that do not. To save including this overhead in the vast majority of C programs that do not use floating point, the standard C library LIBC_A only contains versions of these routines which do not support floating point.

To use the versions that do support floating point you need to include the maths library LIBM_A by specifying the -lm parameter to either CC or LD. This will cause the floating point variants of the above routines to be included in preference to the integer only ones in LIBC_A.

If you attempt to print any floating point numbers and you have forgotten to include this library then the message

```
    "No floating point"
```

will be printed where the number would otherwise have been printed.

SIGNAL HANDLING

C68 programs that are compiled with Release 4.15 or later will by default support signal handling as described in the SIGNALS_DOC file.

WILDCARD HANDLING

In a number of places you will find reference to 'wildcard', particularily in reference to filenames. The style of wildcard supported is the same as that in Unix. This means that:

a) The character * is used to represent a string of any characters, and of indeterminate length.

b) The character ? is used to represent a single character of any value.

c) The characters [and] are used to give a range of characters at a given point. These can be individual characters (i.e. [ab] means 'a' or 'b') or a range of characters (i.e.[a-c] means 'a' or 'b' or 'c').

d) If you want one of the special characters used above as an actual character value then it is preceded by \.

To look at some examples to help clarify this:

`*_c`	Any file finishing with the letters _c
`\*_c`	A file name whose exact name is '*_c'.
`[a-z]*_[ch]`	Any file whose name starts with a letter, and which finishes with either '_c' or '_h'.
`???_c`	Any file whose name is exactly 5 letters long, and whose last two letters are '_c'.

There are routines in the supplied LIBC_A that support this form of wild card, so it is easy to implement it in your own programs.

WILDCARDS IN COMMAND LINES

A situation in which you commonly want to use wildcards is when passing filenames as parameters to a C program. It is possible to get C68 to emulate the Unix shell capability whereby the command line is scanned for any arguments that contain wildcards, and if any are found they are expanded into a list of matching filenames. This can be achieved in C68 by including lines of the form

```
void (*_cmdwildcard)() = cmdexpand;
```

somewhere in your program outside any function declaration. If you wish alternative wildcard expansion to that provided then you can use your own routine in place of the supplied cmdexpand() routine (but use the source of the supplied one as your model).

For details on how to write such routines it is best to examine the source code for the cmdexpand() routine which is present on the SOURCE issue disks (in the LIBC_INIT_src_zip archive).

USING THE GST LINKER WITH C68

The output '_o' files that are produced by C68 are in standard QL SROFF (Sinclair Relocatable Object File Format). They are thus acceptable to the the standard GST LINK program (or the Qunata QLINK program which is just a bug-fixed version of LINK). LINK writes its relocatable information in a different format to the LD linker, which means that it is necessary to use a different startup module to the ones provided for use with LD. A suitable start up module for EXECable programs is provided as 'qlstart_o' with the C68 system.

Note that the GST LINK program is not capable of linking RLLs, or of linking programs that will use RLLs. For this you must use the LD linker (v2.00 or later).

It is assumed that if you intend to use LINK, then you know how to run it, so no additional intructions are included in this document.

CC THE C COMPILER FRONT END

CC - C compiler front-end.

SYNOPSIS

CC [options] filelist

DESCRIPTION

The CC command is the users command-line front-end to the compilation system. It provides a convenient method of controlling and running all the underlying components. The CC command can support either the "C68 system for QDOS" or the "CPOC system for the Psion 3a".

In the description of the options, the program names in square brackets show which of the underlying compilation phases use any particular option. For more detailed descriptions of the options that are not specific to CC only, refer to the documentation specific to the underlying programs mentioned in the square brackets.

The CC front end uses the extension part of the filename to decide which phases are appropriate to any particular filename. It is important therefore that you stick to the filename conventions laid out later in this document.

The compile options are preceeded with a '-' to differentiate them from any source file name. Note that case is significant when specifying options unless indicated otherwise. Options can alternatively be taken from Environment Variables (as detailed later).

`-A` `[AS68]`
See AS68 documentation

`-bufl` `[LD]`
Change the buffer length for reading libraries. See the LD documentation for nmore details.

`-bufp` `[LD]`
Change the buffer size for holding the final binary program. See the LD documentation for more details.

`-c` `[CC]`
Stop after the assembler phase. This will produce an object file suitable for input into the linker. This is the option used when you are compiling individual modules that will later be linked together.

`-crf` `[LD]`
See LD documentation for details.

`-C` `[CPP]`
Do not discard comments, but pass them through to the main compile phase.

`-d` `[CPP]`
See CPP documentation for details

`-D` `[CPP]`
Pass "defines" to the pre-processor.

`-DEBUG` `[LD]`
See LD docuementation for details

`-error=n` `[C68/C86]`
Set the error level.

`-extern` `[C68/C86]`
See compiler documentation for details.

`-E` `[CPP]`
See CPP documentation for details

`-format` `[C68]`
`-noformat`
See the C68 documentation for details.

`-frame=n` `[C68]`
Set the frame pointer index register.

`-g` `[C68]`
Produce debugging information. Currently this has little effect in C68. It does, however cancel any -O option if that is also specified.

`-h`
This option is no longer used. It is not passed to any of the underlying programs but is accepted by CC for backwards compatiblity reasons although otherwise ignored.

`-icode` `[C68]`
Output details of the internal code tables.

N.B. This option is only available if C68 was generated with the ICODE option set in its configuration file. The standard version of C68 does NOT support this option.

`-I` `[CPP]`
Specifies search sequence for header files.

This means that it is not necessary to include the pathnames of include files in your source programs. Standard header files on the distribution disk are normally included by the line

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

if they are kept in the include_ sub-directory in the default program directory.

`-llibid` `[LD]`
Specify library(s) to be searched when linking the program before the standard default LIBC_A library. The libid field will have the text "lib" appended to the front, and "_a" at the end to derive the library name. Thus using -lm would result in the library libm_a being searched.

`-lattice` `[C68]`
Allow LATTICE style proto-types to be used.

`-list` `[C68]`
Output a listing file.

`-L` `[LD]`
Specify the directory search sequence for standard libraries to be used in the link.

`-m` `[LD]`
Produce a map file.

`-maxerr=n` `[C68]`
Set the maximum number of errors that should be reported by C68 before abandoning the compilation.

`-ms` `[LD]`
Produce a map file plus sumbol information.

`-M` `[CPP]`
Passed to CPP. Produce output suitable for MAKE describingdependencies.

`-MM` `[CPP]`
Like -M, but system header files not included in list of dependencies.
Passed to CPP

`-nostdinc` `[CPP]`
See CPP documentation for details.

`-N` `[AS68]`
Do not attempt to optimise code. By default AS68 will attempt to use short addressing modes where it can to reduce the size of the code.

`-o file` `[LD]`
Specify name out output program file. If not specified, then a_out will be used.

`-opt` [C68]
`-noopt`

See C68 documentation for details.

`-O` [C68]
Invoke the maximum level of optimisation. This can produce quite a significant reduction in program size as well as normally giving more efficient code, so it is normally well worth doing. A much more detailed discussion of the optimisation process is given in the documentation of the c68 program itself.

`-p` [CC]
Stop after the CPP pre-processor phase. This will produce a file (ending in _i) which has the C source after pre-processing that would normally be input to the C68 phase.

`-pedantic` [C68]
See C68 docuemntation for details.

`-P` [CPP]
Passed to CPP. Inhibits generation of # lines in the output giving line number information relating to the original source file. Needed if assembler is being passed through CPP.

`-qmac` [CC]
This option is no longer used. It is not passed to any of the underlying programs but is accepted by CC for backwards compatiblity reasons although otherwise ignored.

`-Qoption` [C68]
This option is used to pass options to the C68 phase that are not catered for by CC. It is followed immediately by the option you are interested in. For further details see the C68 documentation.

`-rlibid` [LD]
Specify Runtime Link Library (RLL) library(s) to be searched when linking the program before the standard default LIBC_A library.

`-reg` [C68]
`-noreg`

See C68 documentation for details.

`-R` [LD]
Specify the directory search sequence to be used for locating Runtime Link Libraries (RLL's).

`-s name` [LD]
Specify the name of an alternative start-up module from the default value of crt_o.

`-stackcheck [C68]`

See C68 documentation for details.

`-stackopt   [C68]`
`-nostackopt`

See C68 documentation for details.

`-sym      [LD]`

See LD documentation for details.

`-S        [CC]`

Stop after the C68 compilation phase. This will produce a file (ending in _s) which has the assembler source produced by the compiler. Normally this is input into the AS68 assembler phase to produce the object (_o) file.

`-tmp      [CC]`

Specifies the device and/or directory that will be used to hold intermediate files. These are work files created during the compilation process that are deleted on completion. Therefore

```
     -tmp ram1_
```

would cause all temporary files to be put onto ram1_. The default is to use the same device as the input file to the relevant phase.

`-TMP      [CC]`

This option is similar to the -tmp option above, but the final output file (typically the _o file)is also put onto the device specified.

`-trace    [C68]`

See C68 documenation for details.

`-trad     [C68]`

Revert to standard K&R compatibility mode. Disables most ANSI features.

`-trigraphs [CPP]`

Accept trigraphs in the C source.

`-uchar    [CPP, C68]`

Treat the 'char' data type as unsigned. By default it is treated as signed.

`-undef    [CPP]`

Suppress definition of standard pre-defined symbols.

`-unproto`

This options is no longer used. It is not passed to any of the underlying programs but is accepted by CC for backwards compatibility reasons although otherwise ignored.

`-U` `[CPP]`
Forbid defines for the specified symbols. Overrides the -D option if necessary.

`-v` `[CC, CPP, C68, AS68, LD]`
Run in verbose flag. This means that CC displays the command line used to run each phase of the compilation system as it is invoked. This is particularily useful if you are getting a compilation failure and you are not sure at what stage of the compilation process.

The -v flag is also passed to each of the phases that CC is running. This will cause these underlying programs to output a message giving their version number.

`-V` `[CC]`
This is like the -v option in that it causes CC to display the command line used to invoke each underlying program. The difference is that the -v flag is not passed to these underlying programs to make them output their own version number message.

This mode is also invoked automatically if CC is started directly from the command line (as opposed to via some other program such as MAKE or C68MENU), and the -v flag is not present.

`-warn=n` `[C68]`
Set the maximum level of warning reports.

`-x` `[LD]`
Include a external reference symbol table in the final linked program.

`-Xa` `[CC]`
`-Xc` Determines compatibility modes. In particular
`-Xt` This option affects the handling of errors in the maths functions. See the LIBM documentation for details.

`-Ypath` `[CC]`
Set program search path for CC. The default location that is used by the C68 compilation system to look for all system files is the default program directory as set by the Toolkit 2 PROG_USE command. The -Y option allows an alternative device and/or directory to be used as the location for finding all system files used by the various compiler phases.

As an example
```
    -Yflp1_
```
will cause the programs to look for the system files from FLP1_. A directory can also be given. -Yflp1_comp_
will cause the programs to look for the system files in the directory FLP1_COMP_. You can combine these two usages to use sub-directories off the default program directory. Therefore
    Eg. `-Ycprogs_`

will look in the cprogs_ sub-directory of the default program directory.

The -Y option effects all file paths that would otherwise be relative to the default program directory such as the default path for system include files and libraries.

ENVIRONMENT VARIABLES

It can be more convenient to set certain options for CC via Environment Variables rather than via the CC command line. The following Environment Variables are currently supported:

`TMP`	Specifies the device and/or directory that will be used to hold intermediate files. Equivalent to the -tmp parameter line option.
`TEMP`	An alternative name to TMP with the same function. If both are present then the -TMP option takes precedence.

There are then a number of environment variables that allow you to control where components of your system are lcoated. The names start with a prefix dependant on the target system that the front-end has been built to support as follows:

`CC_xxx`	option when front-end has been built to support development of programs for "C68 for QDOS".
`CPOC_xxx`	option when front-end has been built to support development of programs for "CPOC for the psion 3a".

This approach has been taken so that you can have to variants of the front-end co-existing on the same system - one targetted at QDOS and the other at CPOC. The options available (where the 'xx' part indicates the prefix as indicated above are):

`xx_OPTS` This allows any options that would normally be passed via the command line to be preset. The environment variable information is processed before the command line, so in the event of any conflict the command line information will take precedence.

`xx_PATH` The location that is used to hold the programs underlying the compilation system. Equivalent to the -Y command line option.

Defaults:
```
CC_PATH=
CPOC_PATH=
```

`xx_CPP` The name of the C preprocessor to be used.

Defaults:
```
CC_CPP=cpp
CPOC_cpp=cpoc_cpp
```

`xx_COMP` The name of the main C compilation phase that is to be used.

Defaults:
```
CC_COMP=c68
CPOC_COMP=c86
```

`xx_ASM` The name of the assembler to be used.

Defaults:
```
CC_ASM=as68
CPOC_ASM=c86
```

`xx_LD` The name of the linker to be used.

Defaults:
```
CC_LD=ld
CPOC_LD=ld86
```

`xx_INC` The location of include files for use by the pre-processor. If this environment variable is specified, then a -I parameter specifying this path will be automatically generated and passed to the pre-processor.

Defaults:
```
CC_INC=
CPOC_INC=
```

`xx_LIB` The location of library files for use by the linker. If this environemtn variable is specified than a -L parameter specifying this path will be automatically generated by CC and passed to the linker.

Defaults:
```
CC_LIB=
CPOC_LIB=
```

If an option is also specified via the command line, then this overrides the setting of the Environment Variable.

EXIT VALUES

The CC program returns the following error codes:

0	All compilations were successful. That is, at least one source file was compiled, and there were no fatal errors.
1	One or more fatal compilation errors were reported.
2	No source files were found.
< 0	QDOS error code. A problem was encountered in running the compiler driver (eg. No memory).

THE COMPILATION PROCESS

The actual compilation process takes place in several phases. Each phase is performed by a separate program. All these programs are controlled by CC so that the user does not have to run them individually. However, awareness of the process helps understand many of the error conditions that can arise. In particular the filename extensions are used by CC to decide what actions are required for a particular file.

C source files are expected to have the extension 'c'. These files are passed to the pre-processor to produce an 'i' file. The pre-processor phase actions all keywords in the C source file that begin with # symbol.

The next stage is the main C compilation phase in which the C code is analyzed and validated. The input to the compile phase is an 'i' file (or a 'k' file if the -unproto option is used) from the pre-processor stage. The compile phase generates assembler output which is put into a file with a 's' (or 'asm' if one the additional optional compilers is used) extension. You may wish to look at this file to see what code has been generated by your C program.

The C68 version of the compiler an generate assembler code file in two formats. The 's' extension is used if it is in the format used by the AS68 assembler provided with the C68 system. The 'asm' extension is used if it is in a format suitable for use by the QMAC assembler (an enhanced version of the GST macro assembler will be obtainable from the QUANTA User Group). N.B. the version of QMAC currently available is not suitable - an announcement will be made when the enhanced version becomes available.

The assembler file is now compiled down into an object file and put into a file with an 'o' extension. The format of this object file will be SROFF (Sinclair Relocatable Object File Format) for QDOS targets, and the MINIX object format for CPOC.

Finally the users object file(s) are input into the linker that converts them into machine code, and adds support routines from the libraries supplied with C68. The output from the linker is a program that can be run with the EXEC command (or an equivalent) from SuperBasic.

If this process seems complicated do not worry. The CC front-end program takes control of this process so that it is easy to use.

It is also possible to get the CC command to run assembler files through the C pre-processor, and then pass them to the Assembler phase without attempting to run the compilation phase in the middle. If the filename extension is 'x', then this is done automatically. If the filename extension is 's' then the source file is examined, and if the first character is a # symbol, then the pre-processor is run before the assembler (This last action is for comaptibility with tradional Unix treatment of assembler files).

COMPILING A C PROGRAM ON QDOS

We now look at some practical of CC to compile C programs. Note that the C68 compilation system will expects to be able to use use Toolkit 2 directories. This means that TOOLKIT 2 is highly recommended for running the C68 system. Programs generated by the C68 system will use Toolkit 2 directories if present, but will also work satisfactorily without it. However, certain library calls require TOOLKIT 2t, so for programs to work on all QL's these should be avoided. The library documentation will state when routines use TOOLKIT 2.

To compile your program (for example test_c) simply type (from SuperBasic):-

```
EX cc;'test_c'
```

The above command loads the compiler phases from the default program device, and compiles the source file test_c found on the default data device, writing out a file test_p from CPP, replacing it with a file test_s after running C68, and test_o file after running AS68. Finally the linker will produce an output file called a_out.

Any errors or warnings are reported in an on-screen window. You can also get CC to display the command lines for each phase as it is run by including the -v option, and put the final program into a specified file by using the -o option. To do this the above command line becomes

```
EX cc;'-v -otest test_c'
```

The output from the compiler passes may be redirected into a file by use of the UNIX style >, and &> commands. For example, to redirect standard out (the compiler sign on messages) to a file ram1_wombat, you would type

```
>ram1_wombat
```

anywhere in the command line. To redirect stderr as well (the channel used for any fatal QDOS error messages from CC) you would use

```
>&ram1_wombat
```

Finally to append either of the above commands to an existing file without destroying it's contents you would use

```
>>ram1_wombat  in the first instance, and
>>&ram1_wombat in the second.
```

Redirection is covered more fully in the QDOSC68_DOC document.

Wildcards may be used to select the files to be compiled. These follow the Unix rules for wild-cards - see the INTRO_DOC document for more details. For example, to compile all files in the current data directory ending in _c you would use:

```
    EX CC;'<compiler options> *_c'
```

The asterisk tells the CC program that the given name is a wildcard. It will then match any filename element that is before _c. To compile files starting in arc and ending in _c you would use :

```
    EX CC;'<compiler options> ARC*_C'
```

Wherever an asterisk appears CC will try and match a filename element to the name. However, if a name begins or ends with any characters other than an asterisk, then these characters must be matched exactly. Asterisks can also be used within filenames,

```
    eg.  tes*_wom*_c
```

matches test_wombat_c, tester_woman_user_c, but would NOT match the filename test_wom_c_hello. This wildcard matching is the same as that used in the directory access functions described in the library, and so is also available to your own programs. The CC program has a 4K buffer for filenames, so that is the limit on the total length of the names of all the files to be compiled.
KNOWN PROBLEMS

1) If an option name is mispelt, then it may be treated as a different option which has less text and thus not have the expected effect. This could mean that CC does not act as expected or that the option is passed to the wrong program.

As an example, if you typed -cr instead of -crf it would be treated as -c by CC, which would thus stop after creating any _o files without proceeding to the link stage.

Another example might be if you specified -maxerrors instead of -maxerrors would result in the parameter being passed to LD (as though it were -m) instead of to C68 as you probably intended.

C68Menu v4.0

1. INTRODUCTION

C68Menu is a front end to the C68 'C' compiler system. It is designed for those who wish to:

a) Select from a directory listing rather than type in a filename

b) Forget how to start Editors, Compilers, Linkers etc

c) Forget compiler and linker switches like -v -ms etc

d) Add libraries without remembering how to do so, or their names

e) Use make files/alter make files but never have to edit one

f) Use no other fingers than two left thumbs!

2. COMMUNICATING WITH C68Menu

C68Menu is constructed like a form and consists of information boxes and action boxes. These boxes may sometimes produce sub-forms requesting further information. There are two main sub-forms: options and directory.

2.1 MOVING ABOUT THE FORM

Cursor keys move a highlighted cursor round the boxes. Pressing the space bar (or ENTER) will either allow information boxes to be changed or action boxes to 'act'. Action boxes are labelled inside the box in large letters. Information boxes have descriptions outside the box and maybe information inside, both in small letters.

Wherever the cursor is positioned a helpful message is given at the bottom of the screen stating what happens if space or shift-space is pressed. Shift-space sometimes gives access to further less used information and actions.

A quick method of selecting many boxes is achieved by pressing the letter underlined (usually the first) in the name. Pressing shift with this letter is like pressing shift-space with the cursor on that box. C68Menu looks for the shift key to be pressed and ignores CAPS LOCK.

2.2 INFORMATION BOXES REQUIRING EDITING

Selecting some information boxes simply allows editing of the information in the box. These are mainly located in the options and directory sub-forms. Editing is performed in the normal QL way followed by ENTER.

2.3 REQUESTS FOR A SINGLE FILENAME

Selecting the MAKE filename box or EDIT action box will produce a sub-form showing the following infomation boxes: directory, extension and blank filename. Under these is a scrollable list of files in that directory with the given extension. At the bottom is a list of actions you may take.

2.3.1 Changing the directory - the hard way

To change the directory, type 'p' and a cursor will allow you to edit the directory information box.

2.3.2 Filtering out irrelevant extensions

When choosing a file to edit, you may type 'e' and edit the extension box, usually set to _c by default. This filters out all but those files with the given extension, unless left blank in which case all will be shown.

When selecting make files, library files and object files (manual linking only) the extension will be chosen for you and cannot be changed in the directory sub-form.

2.3.3 Entering a new filename

To choose a new filename, type 'f' then you may edit the filename information box. Upon typing ENTER the sub-form will dissapear. Entering a blank filename will not exit the sub-form.

2.3.4 Selecting an existing filename

If you see the file you wish to load in the list of files, simply move the cursor to it using the up/down cursor keys and press ENTER. If you are not in the correct directory, you may change the directory as given above but there is an easier way. Subdirectories are shown as a filename ending in a space then ->. If you select these and press enter, the directory information box will be amended and an updated list of files shown. Similarly, to go up a directory, simply select [parent directory] ->. By this means you may navigate right back to a list of default devices (win, flp1_ to flp4_, ram1_ and ram2_) and back through another device.

This system was designed for 'hard' directories (used on winchester drives and Gold-card (created using MAKE_DIR). For those without this facility, selecting Suppress Dir in the options form will do nearly the same thing. More detail about this is given later.

If escape is pressed at any time, you will exit without a filename being chosen. This will most likely abort any action requiring the information.

2.4 INFORMATION BOXES REQUIRING A SELECTION OF FILENAMES

Selecting the source filenames box and libraries box will produce a similar sub-form. This consists of a directory name, extension, an upper scrollable list of possible filenames, a lower scrollable list of chosen filenames and a list of possible commands. Selecting the directory, extension and navigating around the files is as mentioned above. To select a file to be included in the chosen filenames box, select it with the cursor, then press 'a' (for add). This can be done as many times as you wish. To remove files in this chosen list, press TAB to switch to the lower list of chosen files, select the required one and press 'r'. TAB will return you to the upper list. ESCape will exit. You cannot select files that do not yet exist.

2.5 ACTION BOXES

C68Menu has four boxes labeled EDIT, MAKE, EXEC and OPTIONS. Selecting these will initiate the appropriate action. A fifth box labeled QUIT has the obvious effect.

2.6 SHIFT-SPACE

Shift-space when positioned on the EXEC action box allows parameters to be passed to an executed program.

Shift-space when position on AUTO-MAKE allows just the regeneration of a make file without the make file being run.

3. CONFIGURING YOUR SYSTEM

3.1 Setting up your System Disk

If you have a system that only has 720Kb floppy disk drives, then it is recommended that the C68Menu program is put on the disk you use at BOOT time. As issued, C68Menu is therefore supplied on the RUNTIME 2 disk.

If you have High Density Floppy disks (1.44 Mb) or a hard disk, then it is recommeneded that C68Menu be located alongside the other C68 programs (such as CC, C68, LD etc). In addition, you need to ensure that the file 'touch' is in the same directory as CC. Place the editor of your choice (eg QED) on this disk if there is room. Those with large disks (winchester or 3.2M) may copy all files to a directory of their choice.

You can make a bit of extra space available on the C68 System Disk. First, however, make a copy of the original disk and work with the copy. On this copy delete all the files that have file names consisting of spaces or descriptive comments. These files are merely present to allow "comments" to be added to the information you get when you directory the disk, and are for documentation purposes only.

3.2 Loading C68Menu

You can now start up C68Menu with

```
      EXEC_W C68Menu
```

or

```
      EXEC C68Menu
```

This last option will require you to use CTRL-C to switch programs unless you are using a multi-tasking front-end such as QRAM. The C68 System Disk contains a BOOT program will start C68Menu up automatically.

3.3 Configuring C68Menu

C68Menu is shipped such that the C68 system is expected to be in the PROG_USE directory (normally set to flp1_) and user programs in the DATA_USE directory (normally set to flp2_). These are easily over-ridden once C68Menu is running.

Press 'o' to select further options:

Notice that the box labelled "C68 System" contains "flp1_". If this is not the correct destination for your system, select this box by pressing SPACE, then change the "flp1_" to what you require, ensuring it ends with an underscore (_).

Under this box is one labelled "C68 Temp". This is used for temporary files such as preprocessor and assembler files. It is suggested that this be left as "ram1_" unless you are very short of memory and wish to use a disk drive.

Similarly, you may select the default extensions for make files and execution files. These are best left as they are.

Ensure the box labelled "Editor" contains the correct location and name of your editor (eg flp1_qed).

The options for compiler and linker need not normally be changed.

The option "Suppress dir" should be left set to "Yes" for the moment.

You may set the colours if your monitor/TV does not show them clearly. This is described later.

Once you are happy with the settings, select SAVE. This will modify the C68Menu executable file with your information. If you get an error message saying it could not find C68Menu then you probably have the C68 System directory incorrectly set.

4. NORMAL OPERATION

4.1 Choose a make file

Before you do anything, you must choose a make-file name. You may not know the first thing about make files and possibly do not even wish to use one, but fear not, just give C68Menu a name and it will be magically produced for you without any further ado. Selecting the make file information box will produce a directory sub-form where you can type 'f' and enter the filename. This is entered without an extension (eg prog1). The same name but with the 'exe' extension will be the name of the executable file.

```
*****************************************************
** A bug in the current release may incorrectly put up **
** <alien format> against this filename - ignore this. **
*****************************************************
```

4.2 Create/edit your source files

Strictly speaking you can edit your source files at any time. Press 'e', type in a new filename and press ENTER. The editor chosen will then be run and will automatically read the selected file. When you have finished editing, exiting the editor (F3 followed by X ENTER in the case of QED) will return you to C68Menu. Repeat this process for one source file or many source files. It is normal to have all your source files in the same directory as the make file (which is the default directory that will be selected - if you selected a make file first), although this is not mandatory. Indeed you may require general purpose source files to be in another directory.

Header files may be created and will require the Extension box being selected as _h.

4.3 Informing C68Menu about the source files

C68Menu needs to know which source files are to be included in the final program. Often short programs contain only one. Pressing 's' will produce a sub-form that will allow you to select one or more source files. When finished the main form will re-appear and the first few filenames that fit are listed in the information box.

4.4 Selecting Additional Libraries

By default, the standard library libc_a is always searched. At release 2.00 of C68 this included most C functions and QDOS specific ones. If floating point numbers are used in your program then libm_a needs to be selected. This is selected in a similar way to source files. You will find libc_a in the list of libraries; you never need to select this.

4.5 Selecting AUTO-MAKE

All there is left to do now is to tell C68Menu to GO! Auto-Make does this. Rather than remembering which source files need re-compiling when changes are made and re-linking, C68Menu automatically creates a make file from the information you have given and passes that make file to the make utility which determines all these administrative matters.

4.6 Executing the program

Assuming no errors were reported, typing 'x' will run the program. If you need to pass parameters to it, then shift 'x' will allow these to be set first.

4.7 What went wrong?

Compiler errors usually give reference to line numbers which can be checked by re-editing the source.

Linker errors are sometimes more difficult, for example unresolved symbols are only listed in the map file. To view the map file simply select Edit and set the Extension to _map, then select the appropriate file.

4.8 Re-making

If you have only edited source files, selecting Auto-MAKE will simply run make again with the make file already created. If source files and/or libraries have been added or withdrawn (or certain options changed) then auto-MAKE will re-create the make file then call make. When a make file is generated, it searches all your source files for #includes to headers and #includes in the headers etc in order to determine the dependencies required by the make file. If you change any calls to headers then shift auto-MAKE should be run to force a new make file to be generated (which will re-read the source files). Note, however that other modifications to header files other than #includes do not require this shifted operation.
This requirement saves C68Menu reading every source file every time you auto-MAKE.

4.9 Re-loading the make file at a later date

On starting C68Menu, select 'm'. After selecting the correct directory, your make file should be shown in the list. Selecting this and pressing ENTER will now not only fill the make file information box but you will find that the source files and libraries information boxes will be filled. You may now type 'x' to execute, or edit source files and auto-MAKE, or remove/add libraries and source files. C68Menu has effectively read the make file that you produced last time and extracted all the relevant information.

5. OPTIONS

This menu is rarely required and can be exited with escape.

5.1 C68 System files

720 k floppy disk users will probably have this set to flp1_; however, if you have placed all C68 files (including touch) in a directory for tidyiess, change this. This directory need not be a 'hard' type but could be say flp1_C68_ containing files like flp1_C68_cc, flp1_C68_LIB_libc_a etc.

5.2 Extensions

As C68Menu uses make files to store information on how to make different programs, it was thought wise not to use the convention of calling all make files 'makefile'! If you object to _mak as an extension or want no extension then you may change this.

Similarly C68Menu appends _exe to the root of the filename chosen, rather than calling all executable filenames a_out.

5.3 Temporary files

The "C68 temp" location is where pre-processed and assembler files are put. Normally this is ram1_ which speeds up compilation.

5.4 Compiler/linker/maker options

These are set on shipping to reveal some of the diagnostic information that may be useful. This may be removed. For example, removal of the -ms in the linker options will speed up the operation by avoiding the creating of a map file each time. You may require to increase the buffer size, or wish to halt the compiler after the pre-processor stage etc.

Note that these settings (along with execution file extension) are saved in the make file. Thus beware that they may change when a make file is loaded. This allows some make files to have personal buffer and heap sizes.

5.5 Editor location

This gives the full directory and name of the editor. This is to allow another editor that may not fit only the same disk as the C68 system to be used. Note that when the C68 system is automatically copied to ram disk (see later) this filename will be automatically changed if the editor is also copied.

5.6 Default

This simply undoes all the changes THIS SESSION. Once you save to the C68Menu execution file and exit, you cannot undo these. Thus you are ill advised to ever save the setting to your original copy of the C68 system.

5.7 Compile

This is for when you wish to just re-compile a particular source file. A directory sub-form will appear and after selecting a file, compilation will take place.

5.8 Link

Link allows you to link object files (that do not have source code available). You must have filled in the make file box since this gives the name of the exec file to be produced. If a source files list already exists on the main form then these will already be chosen and you may navigate about and choose other _o files. After selecting the filenames and pressing escape, linking will take place.

5.9 Copy C68 System

For those who have a large amount of memory to spare (ie Gold Card), it is possible to copy the C68 system across to ram1_ (or elsewhere). You are first asked to verify the directory you wish to copy from, then the one to, then, after confirmation, all the files are copied. If the C68 system directory contains any hard sub-directories, these will NOT be copied across, thus it is best to leave the headers and library files in soft-directories as shipped.

If the chosen editor is copied across then this will automatically be amended so that the ram disk version is used. The C68 system files directory is also amended.

5.10 Copy Data

It is possible to copy selected files from a directory of you choice to ram2_ (or elsewhere). First you are given an opportunity to select the directory to copy from (by default this will be the one containing the last make file selected). Then the directory to (ram2_ by default). You are then given a directory sub-form to select files from that directory. At this point you may not navigate out of this directory and any hard subdirectories will not be copied. After final confirmation, the selected files will be copied across and the Home Dir'try ammended on the main form.

If you attempt to quit the program, you will be warned that you are using ram disk. This same option will allow you to copy selected files back to your original directory on disk.

5.11 Colours

If your monitor poorly displays the default colours, you may alter most of these. The colour changing system is rather primitive, it gives you three lines of letters/numbers

```
< FORM  ><W><LIST ><I>PPS
pbt23siSIpbipbtsiSIpbipip
9777407272779779727470400
```

The top line indicates to what windows the lower lines refer. <FORM> refers to the main form and sub-forms, <W> to the warning window, <LIST> to the list of filenames in the directory sub-form, <I> to the input window and input line, P to the proceed line, S the shadow.

The next line details to what the colour refers: p=paper, b=border, t=main text, 2=heading text, 3=mid text, s=strip, i=ink, S=highlighted strip, I=highlighted ink.

The lowest line gives the QDOS colour number for mode 4 (0/1=black, 2/3=red, 4/5=green, 6/7=white), except that 9 gives a rather sexy maroon stipple. If this last colour is not acceptable, it is advised to change it to green (4).

The effect will be immediate.

5.12 Save

This checks that C68Menu is in the C68 system directory, locates the area that contains the options variables in the code and patches the code. If C68Menu is not in the C68 system directory then, although it can be run, it cannot have options saved to it.

6. MISCELLANEOUS

6.1 Soft Directories

In order to mimic hard-directories the Suppress Dir box in the options menu may be set to Yes (space toggles the setting). For both those with and without hard sub-directories it suppresses soft sub-directories which have the same directory and the given extension (eg cprogs_fred_c, cprogs_bob_c) to a single directory (eg cprogs_ ->) and removes all sub-directories which do not possess a file with the appropriate extension. This only works when an extension is given. If you select such a sub-directory and press ENTER then the directory name will change appropriately and a new listing of files in that sub-directory given. Naturally it cannot tell the difference between sub-directory names which contain and underscore and a sub-directory in a sub-directory.

Where the extension box is left blank, this suppression mechanism will not function as it is impossible to determine what is a file with an extension and what is a sub-directory followed by a filename without an extension. The final filename may start with one or more underscores without any problem.

6.2 DATA_USE and PROG_USE

When shipped, C68Menu will use the DATA_USE directory as the default make file directory and PROG_USE directory as the default C68 system directory. This is shown by placing the directory name (only on the main and options forms) in angle brackets (eg <flp2_cprogs_>). If you save (under the options form) then if either directory name is in angle brackets then the appropriate DATA/PROG_USE will be used. If, however, you save when not in angle brackets, then that specific names directory will be taken as the default.

The DATA_USE and PROG_USE directories as set outside this program, although temporarily altered when the program performs some operations, will be left intact when the program is exited.

6.3 Alien Make Files

If you select a make file that has not been generated by C68Menu, then the other information boxes will not be filled and the comment (alien) will be alongside the make filename. If you attempt to auto-MAKE, then you will be warned but allowed to overwrite this make file.

6.4 Windows

If the warning message window covers up some useful diagnostic information, then pressing ENTER and holding down will remove the window but not return back to the form until you release the ENTER key.

If you are multi-tasking with other programs and are using Qram then you should have no problem with destructive windows. If this is not the case, then pressing F4 will refresh when in the main or options window.

6.5 Help

A brief two page set of help notes appear when F1 is pressed. These are just for those who never have time to read documentation (good programs should not need any!).

6.6 Restrictions

If you specify libraries that are in your make-file directory (along with source code etc), auto-MAKE will generate the relevent -L and -l<libname> options for the linker. Because a disconnected list of directories and list of library names is not sufficient in itself to give the user a list of library files when a make file is loaded, the list of filenames is also added to the make file as a sort of comment line. This works fine until you move the whole make file directory to another location. When the newly positioned make file is read in, C68Menu cannot tell it has moved and the list of library files will still refer to the old directory. The make file, however, will work until then next regeneration of the auto-

MAKE file, when the directory of the original directory will be instigated. This may cause problems when copying to ram disk.

6.7 History and possibly a future

The author is not very proud of the state of the C68Menu code! A few months back, feeling deficient in his knowledge of 'C' at work, he decided to obtain C68 and experiment on his QL at home. To his horror he discovered Unix! Urgggh! As he had been pampered by MicroSoft Windows and QuickBasic on a PC, something had to be done about this situation before experimenting with numerous 'C' programs, that would likely take many iterations to get even past the compiler let alone work. So out of this requirement was born C68Menu. This involved many iterations and helpful comments after trying out early versions, particularly from Dave Walker and Colin Horsman.

C68Menu started as a small beautiful program to attempt to research into a suitable user interface for the system. The philosophy was: Get the interface right then optimise the code. No expense (programming time, memory and speed) has been spared to get the user interface as optimised as possible for the 'C' user, however the code never got optimised nor, more importantly, written in 'C' as this would probably avoid it clogging up 10% of the main C68 disk.

I dare any brave fellow (especially those purists who believe it sacrilege to write in SuperBasic) to attempt to re-write the program in a more elegant form. Please contact me if you need help.

C R Johnson
11, The Copse
Tadley
Basingstoke
Hants
RG26 6HX