sinclair

QL Software

PDQC User Guide

Introduction

This User Guide does not try to teach the C language but to describe the implementation used on the Sinclair QL.

The PDQC program is a translator that allows ATARI Lattice C 3.04 to be run on a QL. This is possible as the object file formats produced by ATARI Lattice C are compatible with the GST linker commonly used on the QL.

Compiling a program

C source files on the QL are expected to have the extension _c, the object files produced by the compiler have the extension _o (_q for intermediate files, _p for output from the pre-processor). To compile a program (for example test_c) simply type (from SuperBasic):-

```
EX cc;'test'
```

Note that this program uses Tony Tebby's QL Toolkit 2 extensively and this is required to run it, although not necessarily the programs it produces (certain library calls require it though, so for programs to work on all QLs these should be avoided. These calls are listed in the library descriptions).

The above command loads the compiler passes one and two in turn from the default program device, and compiles the source file found on the default data device, writing out a file test_q after pass one, then replacing it with test_o after pass 2. The progress of the compilation, along with any errors, warnings etc, is reported in an on-screen window.

The full description of the compilation command is:-

```
EX (or EW) cc;'[compiler options] source_file [optional other files]'
```

In addition the text output from the compiler (not the object file output from the compilation) may be redirected to any open SuperBasic channel via the toolkit 2 procedure of preceding the command line options with #n, where n is the desired channel number.

The compiler options are preceded with a -, to differentiate them from any source file name, and must all precede the source file name(s).

Compiler Options

-b

This option causes the compiler to use a base-relative form of addressing when referring to data section locations. This is a more compact instruction format that uses a 16-bit offset field instead of an absolute 32-bit address. The data is addressed relative to the contents of an address register (either a5 or a6, see option -f). This method has the disadvantage of only allowing 64K bytes of data that can be addressed using this option.

The default is to use 32-bit addressing, which allows a data section of virtually unlimited size that can reside anywhere in memory. In addition this option requires that all libraries and object files linked with an object file compiled with this option have also been compiled with this flag. This is because the register used as an offset for the data area is used by the compiler for register pointer variables if this option is not specified. Thus any code used from these modules would corrupt the base access register.

It may be possible to link with modules not compiled with this option if they do not use any register pointer variables but the practice is not recommended. Also the supplied QL library qlib_l has not been compiled with this option, as it would require a special linker to use programs with over 64K of data. This option is still useable if you only wish to link to similarly compiled object files or to your own, carefully designed assembly language object files.

-c

The compiler defaults to a standard close to (but not 100% compatible with) ANSI C, but the -c (compatibility option) can be used to activate some deviations from the standard, and in that way achieve compatibility with previous Lattice C versions. The -c must be immediately followed with one or more letters from the following list, in any order.

c	Allow comments to be nested
d	Allows $ character to be used in identifiers
m	Allows the use of multiple character constants, Eg. 'ab'.
s	Causes the compiler to generate a single copy of identical string constants.
t	Enables warning messages for structure and union tags that are used without being defined. For Example, struct XYZ *p; would not normally produce a warning message if the structure tag XYZ were not defined.
u	Forces all "char" definitions to be treated as "unsigned char".
w	Inhibits warning messages generated for "return" statements that do not specify a return value within an 'int' function. By the ANSI standard such functions should be declared as 'void' instead of 'int'.

-d

This option has two uses. When there is no variable field following the -d, it activates the debugging mode. Additional information is placed into the object module to facilitate symbolic debugging. Also the pre-processor symbol DEBUG is defined so any debugging statements in the source file will be compiled if surrounded by

```
#ifdef DEBUG
... statements
....
#endif
```

Note. The QL GST linker and no symbolic debuggers (to my knowledge) support this extra debugging information. Therefore its use is not recommended.

The -d option can also be used to define preprocessor symbols in the following ways:

-dSYMBOL	Causes SYMBOL to be defined as if you had included the statement #define SYMBOL in your source file.
-dSYMBOL=value	Causes SYMBOL to be defined as if you had included the following statement in your source file. #define SYMBOL value

-e

This option causes the compiler to recognise the extended two byte character set used in Asian applications.
The -e option can be used as follows:

-e, or -e0	Japanese coding is used. Characters 0x81 to 0x9F and 0xE0 to 0xFC are treated as the start of a two byte sequence. 0xA0 to 0xDF are single byte Katakana codes.
-e1	Chinese and Taiwanese coding is used. 0x81 to 0xFC are treated as the start of a two byte sequence.
-e2	Korean character coding is used. 0x81 to 0xFD are treated as the start of a two byte sequence.

-f

This option specifies an address register to use for the stack frame pointer. It should be followed by the number of the address register to be used. Only two values are allowed. -f5 specifies register a5 is to be used, -f6 specifies a6 is to be used. The default register used if -f is not specified is register a6. If the -b option is specified then whichever register is not used as the addressing base register is used as the stack frame pointer.

Note. All the routines in qlib_l have been compiled with a6 as the stack frame pointer, and use a5 as a register variable. Thus use of this option if linking with qlib_l is not recommended.

-g

This option specifies the directory or device from which both passes of the compiler (files p1 and p2 on the distribution disk) and the compiler running program PDQC will be found. Eg. -gflp1_ will cause the above programs to be loaded and run from floppy disk 1. -gflp1_comp_ will cause the programs to be run from the directory comp_ on flp1_. If this option is not specified then the directory used defaults to the default program directory set via toolkit 2.

-h

This option causes the compiler to halt after a fatal compilation error with the message :

```
Compilation failed
Press <ENTER> to exit, any key to continue
```

Pressing enter at this point will cause the compilation to be aborted and no further files (even though specified on the command line) to be compiled. As this option requires a response from the user it is not recommended when output from the compiler is being redirected to a file for later examination, as the compiler will appear to 'hang' waiting for a keypress with no prompt displayed on screen.

-i

This option specifies a directory that the compiler should search when it is attempting to find an include file that is specified in the source file. Eg: using the option -iflp1_headers_ then placing the line

```
#include <test_h> or

#include "test_h" (there is no difference)
```

in your source file, the compiler will first look in the current data directory for the file test_h, and if it is not found the compiler will look in the directory flp1_headers_. If the file is not found in either of these two directories then the compiler will finally look in the directory _include_ . This allows commonly included header files on the distribution disk to be included by the line

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

If they are kept in the include_ directory in the default program directory. Note that changing the directory the compiler is loaded from with the -g option does not change the include directory searched, it is still in the default program directory. As a final help to porting C programs from other systems with different file naming conventions any file ending in .h will be changed to _h before looking for it. So in the above example the file stdio.h would be looked for under the name stdio_h. Up to nine -i options may be used on the command line.

-l

This option causes all objects except for characters, short integers, and structures that contain only characters and short integers to be aligned on longword boundaries, that is, addresses evenly divisible by 4. Structures will be longword aligned if they contain any members that must be aligned.

-n	This option causes the compiler to retain up to 31 characters for all identifiers. The default maximum identifier length is 8 characters. In either case, anything beyond the maximum length is ignored. Care must be taken when using this option when linking with qlib_l, as the routines in qlib_l have not been compiled with this option. Thus if a module compiled with this option makes a call to a library function whose full name is longer than eight characters then the library function will not be found in the link.
-o	This option should be followed by the name of the object file to be produced or the device and directory in which it is to be put.Eg. -otest_o, -oflp1_test_o, or (to just specify the directory to put object files) -oflp1_test_ (NB. When specifying directories, the name must end in a _). It is inadvisable to use this option to specify a complete file name (a name ending in _o) when multiple files are being compiled as all the files would be written with the same name. It is used when a files are to be written into a device or directory which is not the default data device.
-p	Preprocess only. If this option is used pass 2 of the compiler is not run, instead a file consisting of the source file including any preprocessor substitutions and includes is output into a file with an extension of _p, instead of _c.
-q	This option should be followed by the device, directory, or complete file name for any "quad files" which are the intermediate files produced by the first phase of the compiler, and read by pass 2. Examples are:

-qram1_	Put all quad files onto ram disk for speed.
-qtest_	Put all quad files into a directory test below the default data directory.
-qn2_win1_test_	Put all quad files into directory test_ on winchester disk 1 connected to network station number 2.

These quad files are automatically deleted after pass 1.

-r	This option is used to cause the compiler to use PC-relative addresses instead of absolute addresses for external function calls. Target locations must be within +/-32K of the instruction. For small programs this can be very efficient. However in larger programs the target function may not be within the necessary range. This will cause linker errors to occur. To cure these, re-arrange the order of the files submitted to the linker so that all function calls are within range.
-u	This option by itself undefines all preprocessor symbols that are normally pre-defined by the compiler. If you want to undefine a specific symbol it must immediately follow the -u. Eg. -uLATTICE undefines the LATTICE symbol.
-x	This option causes all global data declarations to be treated as externals. This can be useful if you define data in a header file that is included by multiple source files. The -x option can be used with all the files except one, in this case, to cause the data items to be defined in one module and referenced as externals in the others.
-=	This option is followed by a number which sets the size of the stack used by passes one and two of the compiler. Eg. -=10000 gives the compiler 10000 bytes of stack. This would be used when the compilation aborts with an out of stack space error.
-%	This option allows the amount of heap space used by passes one and two of the compiler to be set. Eg. -%50000 gives the compiler 50000 bytes of heap. This option would be used when the compilation aborts with an out of memory error.

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.
QDOS error code (<0)	A problem was encountered in running the compiler driver (eg. No memory).

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 its contents you would use

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

Redirection will be covered more fully later in this document.

Wildcards may be used to select the files to be compiled. These are similar to those in toolkit 2, but slightly more restrictive to allow filenames with common extensions to be used. For example, to compile all files in the current data directory ending in _c you would use:

```
EX CC;',compiler options> __C'
```

The double underscore 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 a double underscore appears CC will try and match a filename element to the name. However, if a name begins or ends with any characters other than a double underscore, then these characters must be matched exactly. Double underscores 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 it has been asked to compile. The compiler driver treats all underscores within names as directory elements that are stripped from the name before producing an output file. As a result, compiling a file test_file_c with the line :

```
EX CC;'test_file'
```

will produce an output file of file_o in the current directory. As far as the compiler driver was concerned, you asked it to compile file file_c in the directory test relative to the current data directory. To get the output file name you require use the -o option, eg. -otest_file_o will produce the correct output.

The C Program Environment

The rest of the document describes how the particular implementation of C has been adapted to use the features of the QL, and to retain as much UNIX functionality as possible.

When a C program is loaded into memory and starts, it first relocates itself to run at the load address, then creates a subsidiary QDOS job, owned by the C program, to act as it's data and stack area. This job is never activated and has no name. The stack area of the C program grows downwards from the top of this area, the heap grows upwards from the bottom. There are checks when memory is allocated from the heap using the sbrk and lsbrk calls to check that it doesn't collide with the stack pointer (it leaves a 'chicken factor' of 256 bytes between the two). However, there is no check made when the stack expands downwards, with the result being that programs that use a lot of stack space may collide with the heap and cause crashes. This system of memory management was chosen (rather than using the QDOS functions directly, or allowing each job a fixed stack and heap area decided at link time) as it allows the stack and heap requirements of the program to be decided by the user at run time, via the % and = command line arguments. The QDOS memory management functions were not used to keep the library as compatible as possible with other versions of lattice. You are free to re-write the memory allocation routines to use QDOS if you wish, all the required QDOS routines (mt_alchp, mt_free, mt_alloc, etc.) are in the library qlib_l.

The user may set the memory allocated to both stack and heap at run time by including the command line arguments =ssss, and %hhhh where ssss is a decimal number denoting the amount of stack given to the program, and hhhh is a decimal number denoting the amount of heap (four digits are used for illustration purposes only, in reality the only limit is the amount of memory in the machine). If neither of these arguments are used then the programmer can set the default values of stack and heap that will be allocated at run time by including the variables _STACK and _MNEED in any of his source files. Eg. To set 20K heap space, and 5K stack space then add the lines :-

```
long _STACK = 5L*1024L;
long _MNEED = 20L*1024L;
```

outside any function declaration (to ensure the variables _STACK and _MNEED are global in scope). If _MNEED is set to a negative value then the amount of heap given to the program will be the total memory space in the machine MINUS the absolute value of _MNEED. You needn't declare these values in simple programs where you don't much care what values stack and heap have, the default values declared in qlib_l (8K heap, 2K stack) will cope with most undemanding programs. The facility to set the memory requirements is provided for those programs that need it.

After creating the memory job the program copies its name into the first part of the program space. This is so that QL job listing utilities can display a sensible name for the job. To set your programs name just declare a name by including the line

```
char _PROG_NAME[] = "My_program_name";
```

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

```
char *_PROG_NAME = "My_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 needn't be defined if you don't mind your program being called C_PROG.

After copying the name 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 are put into the array from the command line, each argument being separated from the others by one or more space characters. Note that 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. Eg 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 preceding them with a \ character. A \ character may itself be included by using \\. Eg. 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
```

The special characters = and %, used to denote memory sizes, are not copied into the argument array as they are stripped from the command line to calculate memory size before the command line is parsed.

Similarly, the redirection operators <, >, >&, >>, >>& are also stripped from the command line before parsing. These are the UNIX compatible symbols used to redirect the programs input and output streams from the normal console channel. Their action is as follows:

<filename	This redirects the standard input (file descriptor 0) 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	This redirects the standard output channel (file descriptor 1) 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	This redirects the standard output only 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, no existing data will be overwritten.
>&filename	This command redirects the standard output and standard error (file descriptor 2) channels to the designated file or device. The file is created if it doesn't exist, or truncated if it does. If this command is not present then stderr outputs to the same CON_ channel as stdin and stdout. However, if both of these have been redirected then stderr outputs to a newly opened SCR_ channel.
>>&filename	This redirects stdout and stderr to filename, creating it if it doesn't exist, but opening it for appending if it does.

File descriptors

Standard C communicates to the outside world via file descriptors. These are positive integers between zero and twenty that specify output channels. Obviously they are different from the 32 bit QDOS channel id's. The file descriptors are mimicked under QDOS by having an array of twenty _UFB structures (defined in ios1.h). These structures contain the QDOS channel that the file descriptor that indexes it will output to (or input from). So when a file descriptor is passed to a read call for example, it indexes by file descriptor into the _UFB array, reads the QDOS channel id from it, then does a QDOS read call on this channel. This approach allows C programs to be very UNIX compatible (many UNIX programs will recompile and run without any problems), but also allows the programmer who wants to get the QDOS channel id to do specific QDOS calls to get at the channel easily. It also makes possible library calls such as fcntl, dup, and dup2.

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. However, if the mode is changed to 'raw' (fd opened with O_RAW flag set, or fcntl or iomode call done on 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.

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 fd with the O_NDELAY flag set, or doing a fcntl 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 _UFB structure 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 stat 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.

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 ie- 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 = [reqired size]; /* Outside any function */
```

Running child jobs

The standard qlib_l library has calls that mimic the UNIX fork and exec calls closely, but not exactly, the differences being due to the differences in 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 it's 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 it's child jobs to terminate (an example of wanting to do this would be 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 (suspended jobs do NOT count - so it doesn't detect its own data job for instance), otherwise it waits for one of it's 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 it's 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 QODS priority of _DEF_PRIORITY. This is a global variable in qlib_l, and so may be changed from it's 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 jobs 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 and 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.

To do this the flags field in the _UFB structure (defined in ios1.h) must have the UFB_NC (no close) flag set , and then the fd closed. Setting this flag prevents the actual underlying QDOS channel from being closed (it is in use by the child job) but allows the fd to be reused. as there are only 20 fd slots per job, conserving them in this way can be a good idea. A sample of code to achieve this (it cannot be done via iomode or fcntl calls as this would have made them non-standard) would be:

```
#include <ios1.h>

... Later in a function ....
struct _UFB *uptr = chk_ufb( fd );

uptr->ufbflg |= UFB_NC; /* Set the no close bit */
```

Reading QDOS Directories

There are two 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). All directory reading calls take wildcard parameters (described above in the CC documentation) 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).

The full range of QDOS trap calls are provided as separate routines, including all the graphics calls, which have been expanded to work with both integer and Lattice C double precision floating point arguments. As a rule, the sd_i type routines take integer arguments, whereas the sd_ type routines take double arguments (in graphics traps). Routines have also been provided to convert short and long integers and double precision floating points to QDOS floating point. No routine converts the other way around yet but one is being written and will be added to the library at a later date. 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 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, but to access them it is safer 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. Note that, as supervisor mode uses a different stack to user mode it is VITAL that your program doesn't 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.

Using Floating Point

To save space in the printf etc routines, floating point has been excluded from the qlib_l library. If you wish to use floating point in your programs then uncomment the line in the standard linker control file prog_link that reads the floating point library before qlib_l. Doing this will allow the correect floating point versions of any affected functions to be loaded instead of the ones in qlib_l.

Miracle Systems Winchester Disk

As this software was developed on this product (and under QRAM - QRAM calls are also to be included in the library at a later date), there is a special file wini_h on the distribution disk to cope with the extensions of that product. Also the extension calls are supplied in the qlib_l library (for descriptions of them see the miracle systems documentation and the library calls list file, also wini_h).

Appendix A : The Story of PDQC

As told by Dave Walker on his web site.

A QL software house called PDQL had decided that it would like to bring to the market a new C compiler, and gave it the working name of PDQ C. This was to be a full C development environment right from the editor through to the required runtime libraries and was intended to act as replacement for QLC (a QL version of Lattice C) which by then was no longer being actively marketed or supported on the QL. PDQL started advertising PDQ C product from late 1989. Although this product was never made generally available in the marketplace, it was not completely "vapourware". PDQ C was in fact ready by about December 1989 if licensing problems could have been sorted out.

The work to produce PDQ C had been done by Jeremy Allison. He had done this out of interest, and not as a result of a particular commission from PDQL to produce such a product. The reason that there were problems with the licensing was that the PDQ C product was based on Lattice C. PDQL never proved to the satisfaction of Jeremy Allison that they had the rights to distribute a product containing Lattice C, so the final product was never passed to PDQL.

This port of Lattice C was interesting in that it was done by writing an emulator for the QL that allowed it to emulate part of the Atari ST operating system. The details do not matter too much (if you are interested I have written a separate description of this emulator), but suffice to say that it:

- Was efficient
- Allowed the Atari ST version of Lattice C to run on the QL. This was version 3.04 of Lattice C which is a "bug-cleared" version of the Lattice C v3.02 that Metacomco C was based on (which will be refered to as QLC from now on).

The real improvement that PDQ C was going to have over QLC (and the area needing the most work) was that the C library had been extensively re-written. Some of the implications of this improvement in the C library were:

- It was clear of the bugs that had plagued the QLC implementation.
- The standard C file handling routines were enhanced to support Toolkit 2 directories.
- Full support for all QDOS trap and vector calls.

I got involved late in the project to help with the testing and the last stage of work on the libraries. The development had by now progressed until it was largely completed. Unfortunately, as mentioned above, PDQL never completed the necessary licensing arrangements for Lattice C, and eventually PDQL went into receivership. Jeremy decided to wash his hands of the PDQC development and put the work he had done to date into the Public Domain.

This meant that there was a largely complete C system, but that it depended on a core component - the Lattice C compiler - that was not public domain., and this was where C68 came into the picture. It was a compiler for which the source was publicly available and that could be freely distributed. This could replace the Lattice C component of PDQC. Lattic C also had a built in pre-processor facility, and for this the public domain DECUS pre-processor (later the GNU one replaced DECUS) was ported to do the job. The product was released as "C68 for QDOS". This has become the dominant C compiler within the QDOS marketplace - and in fact a number of people have credited the arrival of C68 on the QDOS scene as being one of the factors that has allowed QDOS to survive as it allowed quality software to easily be ported from many other environments.

The "C68 for QDOS" product has gone through a number of major releases, and minor upgrades between those releases. The idea was that we changed the major version number when there might be object code incompatibilities, and the minor number was incremented for less severe changes. The major releases broke down as follows:

- Release 1.xx series was the initial release. This probably should have been called a beta release as at this stage as although we had a working product there were still many problems. In particular this applied to the libraries as in many cases we had objects to which we did not have sources - somewhere along the line they had got lost! All I can say is that the product was already probably comparable in quality to the commercial offerings and it was free.
- Release 2.xx series was much more robust. By this stage we had source to everything which was a major step forward. We also developed utilities such as a librarian to help with maintenance of object libraries.
- Release 3.xx series was when we switched the floating point support from using the Motorola Fast Floating Point format to using IEEE. Also during this release we upgraded the library support to ensure that as well as all the ANSI and Unix C interfaces, we also provide access from the C level to all the native operating systems calls available in QDOS.
- Release 4.xx series was when we finally got to ANSI compliance. Both the compiler and libraries were bought in step.

The above program was a major driving force for developments in the compiler itself. It gave both an incentive to make improvements, and also an eager user population willing to try things out. The major developments that took place in the compiler program itself during this phase is give below. Some developments were in the front-end syntax analysis phase of the compiler and this became available in all variants with no further work, while others were in the code generation side and thus specific to the target processor (which for this period was mainly the 68000 family):

- Floating point support improved.
- The development of support for IEEE floating point in place of the MFFP support that was in earlier releases of C68. This support has gone through two major phases; the first being to add IEEE support purely within software, and the next (relatively recently) to add the option to generate code that calls on hardware FPU directly.

- Multiple processor support improved.
- You can now specify when you build a version of the compiler that you want support for more than one target processor built in (previously there was a choice, but you could only chose one). To complement this you can specify which one you want as the default, and there are runtime options to pick any alternatives.
- Multiple assembler support improved.
- Rather like the multiple processor support you can build in support for many assemblers; specify the default one for each target processor type, and also specify any of those you added support for by runtime parameters.
- The warning system has dramatically improved. Every time we made a mistake in a program that although syntactically correct may well have not been what the programmer meant (such as use of = instead of ==) we looked at whether the compiler could have warned us about it. After a while we also started looking consciously at the warning messages output by other compilers and by tools such as LINT to see if we could build in similar capabilities. The result is a very comprehensive warning system that has been graduated to allow you to select any level of 'pedanticity' that you like.
- Major improvements in the optimisation. The code generated by the latest QDOS releases can be anything up to 25 per cent smaller than was the case for the first release. It is noticeable that although the compiler has grown dramatically in capability over this period, it has not grown that much in size. In fact one of our criteria for a good optimisation was that it should reduce the size of the existing code in the compiler by enough to offset the additional code that we had to include to implement the new optimisation.

By late 1994 the QDOS release of C68 was stable and all the immediate concerns had been addressed. While work has continued within the QDOS implementation it has tended to focus on aspects of the QDOS development system that do not concern the compiler itself. However any improvements to the compiler are fed back into the QDOS based releases very rapidly as it provides a good basis for us to get early feedback on any new features.

Appendix B: A short note on the TOS emulator for QDOS

by Dave Walker

This product which was mentioned in the history of PDQ C. For those who are not aware of it, TOS is the native operating system of the Atari ST.

It seems ironic that there should be a product that allows QDOS to emulate the Atari ST operating system (at least in part). The QL emulator for the Atari ST is one of the ways forward for those who want to keep running QDOS, but want more powerful hardware. In fact the QDOS emulator on the Atari ST is quite happy to run this TOS emulator (if you can see what I mean)!

The aim of the TOS emulator is to allow TOS binary programs to be run without alteration under the QDOS operating system. This makes a number of programs developed for TOS available in the QDOS environment. The TOS emulator restricts itself to trying to run those programs that are "well-behaved", and do not try to use the graphic capabilities specific to the Atari ST (so unfortunately it cannot run ST games).

The TOS emulator works by intercepting system calls that a TOS program would make on the TOS operating system. As on the QL, TOS programs access operating system facilities by using the TRAP instruction. Under TOS, the traps used are:

```
        #1   GEMDOS calls
        #13  BIOS calls
        #14  XBIOS calls
```

QDOS uses the TRAP values of #0 to #4. This means that the only place there is a conflict is on TRAP #1. To get around this, when it first loads a TOS program, the TOS emulator patches all TRAP #1 instructions to be TRAP #5 instructions instead. It then invokes the QDOS facility that allows TRAP calls from #5 upwards to be re-directed to user supplied routines. The effect of this is that all TOS operating calls made by the TOS program are re-directed to the TOS emulator with virtually zero run-time overhead.

The TOS emulator program then tries to map these TOS system calls onto the appropriate QDOS system calls. This approach does has some limitations however:

- The most obvious one is that if the subject program ever tries to by-pass TOS then it will fail to work as it is not really running under the operating system that it thinks it is.
- A linked problem will be if it tries to access TOS system variables as they will not be there.
- A problem occurs if the TOS program tries to make a TOS system call that the TOS emulator does not support. The Atari ST has facilities that cannot be emulated under a QDOS environment (in particular the GEM graphics interface).
- The ST has the eqivalent to the QL vector calls, and these routines (the LINE A and LINE F routines) are not supported.

However, despite these limitations the TOS emulator does succeed in running a significant proportion of Atari ST programs that do not attempt to use GEM. The most significant of these is the Atari ST version of LATTICE C (version 3.04) which was going to form the basis of PDQ C.

Jeremy Allison (who wrote the TOS emulator) has agreed to put the code for this TOS emulator into the public domain. If anyone is interested in playing with it they can download the TOS Emulator for QL source You will have to be competent in assembler programming to be able to make use of this TOS emulator, so it is not for the faint-hearted. The source is in a format suitable for input to the GST Macro Assembler.

The current version was specifically aimed at supporting those system calls used by the Atari ST Lattice C compiler v3.04, and succeeds in this admirably. More work may well be required to get other programs to work reliably. For instance it does not (yet) successfully run the newer version of Lattice C (version 5) that has been released for the Atari ST. If you do make any improvements, then please pass them back to Dave Walker

Table of Contents