sinclair

QL Software

C68 DOCUMENTATION

Overview

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 perfect 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.

Getting Started

INTRODUCTION

This document is intended to help you get started as rapidly as possible in using C68. Eventually you will need to read the more detailed documentation and may well want to print much of it. This document should allow you to at least get the feel for C68 very rapidly without having to do that much.

It is always a good idea to ensure that you have read the README_DOC file supllied with C68. This contains the issue notes for the current release.

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.

CC Front-End

NAME

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 more 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** documentation for details

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

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

-E [CPP]
See **CPP** docementation 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.

-l libid [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 prototypes 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 symbol 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** documentation 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]
-QMAC
This option 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.

-Q option [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.

-r libid [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** documentation 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 compatiblity 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
-Xt
Determines compatibility modes. In particular This option affects the handling of errors in the maths functions. See the LIBM documenation for details.

-Y path [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

-xxx present then the "-xxx" 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 analysed 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.

CHANGE HISTORY

This section documents major changes that have been made to this document. It's prime purpose is to help those who are upgrading their version of the C68 system to identify what has changed.

31 Dec 93 v4.12 DJW

- Documented fact that CC_OPTS Environment variable is now supported.
- Documented the -V option, and updated the -v description accordingly.
- Updated descriptions of some of the other supported options.

03 Jun 95 v4.25 DJW

Updated documentation to list all options that are currently supported. Re-arranged the list to be in alphabetical order rather than that of the program that uses the parameter option.

Gnu C Pre-Processor

The C Preprocessor

Last revised July 1990

for GCC version 1.38

Richard M. Stallman

NAME

cpp - Gnu C pre-processor

SYNOPSIS

cpp [options] [input_file] [output_file]

DESCRIPTION

Most often when you use the C preprocessor you will not have to invoke it explicitly: the C compiler will do so automatically. However, the preprocessor is sometimes useful individually.

The C preprocessor expects two file names as arguments, infile and outfile . The preprocessor reads infile together with any other files it specifies with '#include'. All the output generated by the combined input files is written in outfile .

Either infile or outfile may be '-', which as infile means to read from standard input and as outfile means to write to standard output. Also, if

outfile or both file names are omitted, the standard output and standard input are used for the omitted file names.

Here is a table of command options accepted by the C preprocessor. Most of them can also be given when compiling a C program; they are passed along automatically to the preprocessor when it is invoked by the compiler.

-P
Inhibit generation of '#'-lines with line-number information in the output from the preprocessor (see section Output). This might be useful when running the preprocessor on something that is not C code and will be sent to a program which might be confused by the '#'-lines

-C
Do not discard comments: pass them through to the output file. Comments appearing in arguments of a macro call will be copied to the output before the expansion of the macro call.

-trigraphs
Process ANSI standard trigraph sequences. These are three-character sequences, all starting with '??', that are defined by ANSI C to stand for single characters. For example, '??/' stands for '\', so ''??/n'' is a character constant for a newline. Strictly speaking, the GNU C preprocessor does not support all programs in ANSI Standard C unless '-trigraphs' is used, but if you ever notice the difference it will be with relief.

You don't want to know any more about trigraphs.

-pedantic
Issue warnings required by the ANSI C standard in certain cases such as when text other than a comment follows '#else' or '#endif'.

-I directory
Add the directory directory to the end of the list of directories to be searched for header files (see section Include Syntax). This can be used to override a system header file, substituting your own version, since these directories are searched before the system header file directories. If you use more than one '-I' option, the directories are scanned in left-to-right order; the standard system directories come after.

-I-
Any directories specified with '-I' options before the '-I-' option are searched only for the case of '#include "file "'; they are not searched for '#include <file >'.
If additional directories are specified with '-I' options after the '-I-', these directories are searched for all '#include' directives.
In addition, the '-I-' option inhibits the use of the current directory as the first search directory for '#include "file "'. Therefore, the current directory is searched only if it is requested explicitly with '-I.'. Specifying both '-I-' and '-I.' allows you to control precisely which directories are searched before the current one and which are searched after.

-nostdinc
Do not search the standard system directories for header files. Only the directories you have specified with '-I' options (and the current directory, if appropriate) are searched.

-D name
Predefine name as a macro, with definition '1'.

-D name=definition
Predefine name as a macro, with definition definition . There are no restrictions on the contents of definition , but if you are invoking the preprocessor from a shell or shell-like program you may need to use the shell's quoting syntax to protect characters such as spaces that have a meaning in the shell syntax.

-U name
Do not predefine name . If both '-U' and '-D' are specified for one name, the '-U' beats the '-D' and the name is not predefined.

-undef
Do not predefine any nonstandard macros.

-d
Instead of outputting the result of preprocessing, output a list of '#define' commands for all the macros defined during the execution of the preprocessor.

-M
Instead of outputting the result of preprocessing, output a rule suitable for make describing the dependencies of the main source file. The preprocessor outputs one make rule containing the object file name for that source file, a colon, and the names of all the included files. If there are many included files then the rule is split into several

If there are many included files then the rule is split into several lines using '\'-newline.
This feature is used in automatic updating of makefiles.

-MM

Like '-M' but mention only the files included with '#include "file "'. System header files included with '#include <file >' are omitted.

-i file

Process file as input, discarding the resulting output, before processing the regular input file. Because the output generated from file is discarded, the only effect of '-i file ' is to make the macros defined in file available for use in the main input.

2.0 QDOS SPECIFIC INFORMATION

This section contains details relating to the QDOS port of cpp. It contains information supplementary to the rest of the manual, and is therefore not exhaustive.

2.1 Predefined Symbols.

The following symbols are predefined under QDOS.

2.1.1 Non-standard Symbols.

MC68000

QDOS

C68

The user should be aware that using the '-ansi' flag will cause these symbols to be undefined. Further information is given in sections 2.1.2 and 3.4.3.2.

2.1.2 Standard Symbols.

__MC68000__

__QDOS__

__C68__

__STDC__

The first three of these symbols are derived from the non-standard predefined symbols above. The last of these indicates whether the compiler is ANSI compilant or not. Since C68 is ANSI compliant, this has the value 1.

All other standard symbols are listed in the remainder part of this manual.

2.2 Filenames.

Filenames given to the #include directive may contain a '.' as the extension separator. During processing, any '.' will be translated to an underscore, and the resultant filename used in subsequent actions. Similarily, any '/' (UNIX) or '\' (MSDOS) directory separators will be translated to underscores.

A filename given to the pre-processor via a #line directive, as may be generated by utilities such as YACC and LEX, will not cause the directory search order to alter.

2.3 Directory names.

Any directory name used in directives, or on the command line should end with the trailing '_'.

2.4 Invoking cpp.

Normally cpp will be automatically invoked via the cc command. This may put limitations on the flags which may be passed to the preprocessor. Fpr details of which cpp flags are supported by cc, please refer to the documentation for cc..

2.5 Environment Variables

In view of the restrictions on invocation caused by cc, an alternative mechanism has been provided via the environment variable CPP_OPTS. The contents of the CPP_OPTS environment variable will be prepended to the supplied command line at execution time, and should therefore contain options in the same form as listed elsewhere in this manual.

3. The C Preprocessor

The C preprocessor is a macro processor that is used automatically by the C compiler to transform your program before actual compilation. It is called a macro processor because it allows you to define macros, which are brief abbreviations for longer constructs.

The C preprocessor provides four separate facilities that you can use as you see fit:

- Inclusion of header files. These are files of declarations that can be

substituted into your program.

- Macro expansion. You can define macros, which are abbreviations for arbitrary fragments of C code, and then the C preprocessor will replace the macros with their definitions throughout the program.
- Conditional compilation. Using special preprocessor commands, you can include or exclude parts of the program according to various conditions.
- Line control. If you use a program to combine or rearrange source files into an intermediate file which is then compiled, you can use line control to inform the compiler of where each source line originally came from.

C preprocessors vary in some details. This manual discusses the GNU C preprocessor, the C Compatible Compiler Preprocessor. The GNU C preprocessor provides a superset of the features of ANSI Standard C.

ANSI Standard C requires the rejection of many harmless constructs commonly used by today's C programs. Such incompatibility would be inconvenient for users, so the GNU C preprocessor is configured to accept these constructs by default. Strictly speaking, to get ANSI Standard C, you must use the options '-trigraphs', '-undef' and '-pedantic', but in practice the consequences of having strict ANSI Standard C make it undesirable to do this. See section Invocation.

3.1. Transformations Made Globally

Most C preprocessor features are inactive unless you give specific commands to request their use. (Preprocessor commands are lines starting with '#'; see section Commands). But there are three transformations that the preprocessor always makes on all the input it receives, even in the absence of commands.

- All C comments are replaced with single spaces.
- Backslash-Newline sequences are deleted, no matter where. This feature allows you to break long lines for cosmetic purposes without changing their meaning.
- Predefined macro names are replaced with their expansions (see section Predefined).

The first two transformations are done before nearly all other parsing and before preprocessor commands are recognized. Thus, for example, you can split a line cosmetically with Backslash-Newline anywhere (except when trigraphs are in use; see below).

```
/*
*/ # /*
*/ defi\
ne FO\
O 10\
20
```

is equivalent into '#define FOO 1020'. You can split even an escape sequence with Backslash-Newline. For example, you can split "foo\bar" between the '\' and the 'b' to get

```
"foo\\
bar"
```

This behavior is unclean: in all other contexts, a Backslash can be inserted in a string constant as an ordinary character by writing a double Backslash, and this creates an exception. But the ANSI C standard requires it. (Strict ANSI C does not allow Newlines in string constants, so they do not consider this a problem.)

But there are a few exceptions to all three transformations.

- C comments and predefined macro names are not recognized inside a '#include' command in which the file name is delimited with '<' and '>'.
- C comments and predefined macro names are never recognized within a character or string constant. (Strictly speaking, this is the rule, not an exception, but it is worth noting here anyway.)
- Backslash-Newline may not safely be used within an ANSI ''trigraph''. Trigraphs are converted before Backslash-Newline is deleted. If you write what looks like a trigraph with a Backslash-Newline inside, the Backslash-Newline is deleted as usual, but it is then too late to

Backslash-Newline is deleted as usual, but it is then too late to recognize the trigraph.

- This exception is relevant only if you use the '-trigraphs' option to enable trigraph processing. See section Invocation.

3.2. Preprocessor Commands

Most preprocessor features are active only if you use preprocessor commands to request their use.

Preprocessor commands are lines in your program that start with '#'. The '#' is followed by an identifier that is the command name . For example, '#define' is the command that defines a macro. Whitespace is also allowed before and after the '#'.

The set of valid command names is fixed. Programs cannot define new preprocessor commands.

Some command names require arguments; these make up the rest of the command line and must be separated from the command name by whitespace. For example, '#define' must be followed by a macro name and the intended expansion of the macro.

A preprocessor command cannot be more than one line in normal circumstances. It may be split cosmetically with Backslash-Newline, but that has no effect on its meaning. Comments containing Newlines can also divide the command into multiple lines, but the comments are changed to Spaces before the command is interpreted. The only way a significant Newline can occur in a preprocessor command is within a string constant or character constant. Note that most C compilers that might be applied to the output from the preprocessor do not accept string or character constants containing Newlines.

The '#' and the command name cannot come from a macro expansion. For example, if 'foo' is defined as a macro expanding to 'define', that does not make '#foo' a valid preprocessor command.

3.3. Header Files

A header file is a file containing C declarations and macro definitions (see section Macros) to be shared between several source files. You request the use of a header file in your program with the C preprocessor command '#include'.

3.3.1. Uses of Header Files

Header files serve two kinds of purposes.

- System header files declare the interfaces to parts of the operating system. You include them in your program to supply the definitions you need to invoke system calls and libraries.
- Your own header files contain declarations for interfaces between the source files of your program. Each time you have a group of related declarations and macro definitions all or most of which are needed in several different source files, it is a good idea to create a header file for them.

Including a header file produces the same results in C compilation as copying the header file into each source file that needs it. But such copying would be time-consuming and error-prone. With a header file, the related declarations appear in only one place. If they need to be changed, they can be changed in one place, and programs that include the header file will automatically use the new version when next recompiled. The header file eliminates the labour of finding and changing all the copies as well as the risk that a failure to find one copy will result in inconsistencies within a program.

The usual convention is to give header files names that end with '.h'.

3.3.2. The '#include' Command

Both user and system header files are included using the preprocessor command '#include'. It has three variants:

#include <file>

This variant is used for system header files. It searches for a file named file in a list of directories specified by you, then in a standard list of system directories. You specify directories to search for header files with the command option '-I' (see section Invocation). The option '-nostdinc' inhibits searching the standard system directories; in this case only the directories you specify are searched.

The parsing of this form of '#include' is slightly special because comments are not recognized within the '<...>'. Thus, in '#include <x/*y>' the '/*' does not start a comment and the command specifies inclusion of a system header file named 'x/*y'. Of course, a header file with such a name is unlikely to exist on Unix, where shell wildcard features would make it hard

unlikely to exist on Unix, where shell wildcard features would make it hard to manipulate.

The argument file may not contain a '>' character. It may, however, contain a '<' character.

#include "file"

This variant is used for header files of your own program. It searches for a file named file first in the current directory, then in the same directories used for system header files. The current directory is the directory of the current input file. It is tried first because it is presumed to be the location of the files that the current input file refers to. (If the '-I-' option is used, the special treatment of the current directory is inhibited.)

The argument file may not contain '"' characters. If backslashes occur within file , they are considered ordinary text characters, not escape characters. None of the character escape sequences appropriate to string constants in C are processed. Thus, '#include "x\n\\y"' specifies a filename containing three backslashes. It is not clear why this behavior is ever useful, but the ANSI standard specifies it.

#include anything else

This variant is called a computed #include . Any '#include' command whose argument does not fit the above two forms is a computed include. The text anything else is checked for macro calls, which are expanded (see section Macros). When this is done, the result must fit one of the above two variants.

This feature allows you to define a macro which controls the file name to be used at a later point in the program. One application of this is to allow a site-configuration file for your program to specify the names of the system include files to be used. This can help in porting the program to various operating systems in which the necessary system header files are found in different places.

3.3.3. How '#include' Works

The '#include' command works by directing the C preprocessor to scan the specified file as input before continuing with the rest of the current file. The output from the preprocessor contains the output already generated, followed by the output resulting from the included file, followed by the output that comes from the text after the '#include' command. For example, given two files as follows:

```
/* File program.c */
int x;
#include "header.h"

main ()
{
  printf (test ());
}

/* File header.h */
  char *test ();
```

the output generated by the C preprocessor for 'program.c' as input would be:

```
int x;
char *test ();

main ()
{
  printf (test ());
}
```

Included files are not limited to declarations and macro definitions; they are merely the typical use. Any fragment of a C program can be included from another file. The include file could even contain the beginning of a statement that is concluded in the containing file, or the end of a statement that was started in the including file. However, a comment or a string or character constant may not start in the included file and finish in the including file. An unterminated comment, string constant or character constant in an included file is considered to end (with an error message) at the end of the file.

The line following the '#include' command is always treated as a separate line by the C preprocessor even if the included file lacks a final newline.

3.3.4. Once-Only Include Files

Very often, one header file includes another. It can easily result that a certain header file is included more than once. This may lead to errors, if the header file defines structure types or typedefs, and is certainly wasteful. Therefore, we often wish to prevent multiple inclusion of a

header file.

The standard way to do this is to enclose the entire real contents of the file in a conditional, like this:

```
#ifndef __FILE_FOO_SEEN__

#define __FILE_FOO_SEEN__

...

the entire file

...

#endif /* __FILE_FOO_SEEN__ */
```

The macro __FILE_FOO_SEEN__ indicates that the file has been included once already; its name should begin with '__', and should contain the name of the file to avoid accidental conflicts.

One drawback of this method is that the preprocessor must scan the input file completely in order to determine that all of it is to be ignored. This makes compilation slower. You can avoid the delay by inserting the following command near the beginning of file in addition to the conditionals described above :

```
#pragma once
```

This command tells the GNU C preprocessor to ignore any future commands to include the same file (whichever file the '#pragma' appears in).

You should not rely on '#pragma once' to prevent multiple inclusion of the file. It is just a hint, and a non-standard one at that. Most C compilers will ignore it entirely. For this reason, you still need the conditionals if you want to make certain that the file's contents are not included twice.

Note that '#pragma once' works by file name; if a file has more than one name, it can be included once under each name, even in GNU CC, despite '#pragma once'.

3.4. Macros

A macro is a sort of abbreviation which you can define once and then use later. There are many complicated features associated with macros in the C preprocessor.

3.4.1. Simple Macros

A simple macro is a kind of abbreviation. It is a name which stands for a fragment of code.

Before you can use a macro, you must define it explicitly with the '#define' command. '#define' is followed by the name of the macro and then the code it should be an abbreviation for. For example,

```
#define BUFFER_SIZE 1020
```

defines a macro named 'BUFFER_SIZE' as an abbreviation for the text '1020'. Therefore, if somewhere after this '#define' command there comes a C statement of the form:

```
foo = (char *) xmalloc (BUFFER_SIZE);
```

then the C preprocessor will recognize and expand the macro 'BUFFER_SIZE', resulting in:

```
foo = (char *) xmalloc (1020);
```

the definition must be a single line; however, it may not end in the middle of a multi-line string constant or character constant.

The use of all upper case for macro names is a standard convention. Programs are easier to read when it is possible to tell at a glance which names are macros.

Normally, a macro definition must be a single line, like all C preprocessor commands. (You can split a long macro definition cosmetically with Backslash-Newline.) There is one exception: Newlines can be included in the macro definition if within a string or character constant. By the same token, it is not possible for a macro definition to contain an unbalanced quote character; the definition automatically extends to include the matching quote character that ends the string or character constant. Comments within a macro definition may contain Newlines, which make no difference since the comments are entirely replaced with Spaces regardless of their contents.

Aside from the above, there is no restriction on what can go in a macro body. Parentheses need not balance. The body need not resemble valid C code. (Of course, you might get error messages from the C compiler when you use the macro.)

The C preprocessor scans your program sequentially, so macro definitions take effect at the place you write them. Therefore, the following input to the C preprocessor:

the C preprocessor:

```
foo = X;

#define X 4

bar = X;
```

produces as output:

```
foo = X;

bar = 4;
```

After the preprocessor expands a macro name, the macro's definition body is prepended to the remaining input, and the check for macro calls continues. Therefore, the macro body can contain calls to other macros. For example, after

```
#define BUFSIZE 1020

#define TABLESIZE BUFSIZE
```

the name 'TABLESIZE' when used in the program would go through two stages of expansion, resulting ultimately in '1020'.

This is not at all the same as defining 'TABLESIZE' to be '1020'. The '#define' for 'TABLESIZE' uses exactly the body you specify --- in this case, 'BUFSIZE' --- and does not check to see whether it too is the name of a macro. It's only when you use 'TABLESIZE' that the result of its expansion is checked for more macro names. See section Cascaded Macros.

3.4.2. Macros with Arguments

A simple macro always stands for exactly the same text, each time it is used. Macros can be more flexible when they accept arguments . Arguments are fragments of code that you supply each time the macro is used. These fragments are included in the expansion of the macro according to the directions in the macro definition.

To define a macro that uses arguments, you write a '#define' command with a list of argument names in parentheses after the name of the macro. The argument names may be any valid C identifiers, separated by commas and optionally whitespace. The open-parenthesis must follow the macro name immediately, with no space in between.

For example, here is a macro that computes the minimum of two numeric values, as it is defined in many C programs:

```
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
```

(This is not the best way to define a ''minimum'' macro in GNU C. See section Side Effects, for more information.)

To use a macro that expects arguments, you write the name of the macro followed by a list of actual arguments in parentheses. separated by commas. The number of actual arguments you give must match the number of arguments the macro expects. Examples of use of the macro 'min' include 'min (1, 2)' and 'min (x + 28, *p)'.

The expansion text of the macro depends on the arguments you use. Each of the argument names of the macro is replaced, throughout the macro definition, with the corresponding actual argument. Using the same macro 'min' defined above, 'min (1, 2)' expands into:

```
((1) < (2) ? (1) : (2))
```

where '1' has been substituted for 'X' and '2' for 'Y'.

Likewise, 'min (x + 28, *p)' expands into

```
((x + 28) < (*p) ? (x + 28) : (*p))
```

Parentheses in the actual arguments must balance; a comma within parentheses does not end an argument. However, there is no requirement for brackets or braces to balance; thus, if you want to supply 'array[x = y, x + 1]' as an argument, you must write it as 'array[(x = y, x + 1)]', which is equivalent C code.

After the actual arguments are substituted into the macro body, the entire result is prepended to the remaining input, and the check for macro calls continues. Therefore, the actual arguments can contain calls to other macros, either with or without arguments, or even to the same macro. The macro body can also contain calls to other macros. For example, 'min (min (a, b), c)' expands into:

```
((((a) < (b) ? (a) : (b))) < (c)

? (((a) < (b) ? (a) : (b)))

: (c))
```

(Line breaks shown here for clarity would not actually be generated.)

If you use the macro name followed by something other than an open-parenthesis (after ignoring any spaces, tabs and comments that follow), it is not a call to the macro, and the preprocessor leaves the name unaltered.

Therefore, it is possible for the same name to be a variable or function in your program as well as a macro, and you can choose in each instance whether to refer to the macro (if an actual argument list follows) or the variable or function (if an argument list does not follow).

Such dual use of one name could be confusing and should be avoided except when the two meanings are effectively synonymous: that is, when the name is both a macro and a function and the two have similar effects. You can think of the name simply as a function; use of the name for purposes other than calling it (such as, to take the address) will refer to the function, while calls will expand the macro and generate better but equivalent code. For example, you can use a function named 'min' in the same source file that defines the macro. If you write '&min' with no argument list, you refer to the function. If you write 'min (x, bb)', with an argument list, the macro is expanded. If you write '(min) (a, bb)', where the name 'min' is not followed by an open-parenthesis, the macro is not expanded, so you wind up with a call to the function 'min'.

It is not allowed to define the same name as both a simple macro and a macro with arguments.

In the definition of a macro with arguments, the list of argument names must follow the macro name immediately with no space in between. If there is a space after the macro name, the macro is defined as taking no arguments, and all the rest of the name is taken to be the expansion. The reason for this is that it is often useful to define a macro that takes no arguments and whose definition begins with an identifier in parentheses. This rule about spaces makes it possible for you to do either this:

```
#define FOO(x) - 1 / (x)
```

(which defines 'FOO' to take an argument and expand into minus the reciprocal of that argument) or this:

```
#define  BAR (x) -1 / (x)
```

(which defines 'BAR' to take no argument and always expand into '(x) - 1 / (x)').

Note that the uses of a macro with arguments can have spaces before the left parenthesis; it's the definition where it matters whether there is a space.

3.4.3. Predefined Macros

Several simple macros are predefined. You can use them without giving definitions for them. They fall into two classes: standard macros and system-specific macros.

3.4.3.1. Standard Predefined Macros

The standard predefined macros are available with the same meanings regardless of the machine or operating system on which you are using GNU C. Their names all start and end with double underscores. Those preceding __GNUC__ in this table are standardized by ANSI C; the rest are GNU C extensions.

__FILE__
This macro expands to the name of the current input file, in the form of a C string constant.

__BASE_FILE__
This macro expands to the name of the main input file, in the form of a C string constant. This is the source file that was specified as an argument when the C compiler was invoked.

__LINE__
This macro expands to the current input line number, in the form of a decimal integer constant. While we call it a predefined macro, it's a pretty strange macro, since its ''definition'' changes with each new line of source code.

This and '__FILE__' are useful in generating an error message to report an inconsistency detected by the program; the message can state the source line at which the inconsistency was detected. For example,

```
fprintf(stderr, "Internal error: negative string length"
"%d at %s, line %d.",
length, __FILE__, __LINE__);
```

A '#include' command changes the expansions of '__FILE__' and '__LINE__' to correspond to the included file. At the end of that file, when processing resumes on the input file that contained the '#include' command, the expansions of '__FILE__' and '__LINE__' revert to the values they had before the '#include' (but '__LINE__' is then incremented by one as processing moves to the line after the '#include').

The expansions of both '__FILE__' and '__LINE__' are altered if a '#line' command is used. See section Combining Sources.

__DATE__
This macro expands to a string constant that describes the date on which the preprocessor is being run. The string constant contains eleven characters and looks like '"Jan 29 1987"' or '"Apr 1 1905"'

__TIME__
This macro expands to a string constant that describes the time at which the preprocessor is being run. The string constant contains eight characters and looks like '"23:59:01"'.

__STDC__
This macro expands to the constant 1, to signify that this is ANSI Standard C. (Whether that is actually true depends on what C compiler will operate on the output from the preprocessor.)

__GNUC__
This macro is defined if and only if this is GNU C. This macro is defined only when the entire GNU C compiler is in use; if you invoke the preprocessor directly, '__GNUC__' is undefined.

__STRICT_ANSI__
This macro is defined if and only if the '-ansi' switch was specified when GNU C was invoked. Its definition is the null string. This macro exists primarily to direct certain GNU header files not to define certain traditional Unix constructs which are incompatible with ANSI C.

__VERSION__
This macro expands to a string which describes the version number of GNU C. The string is normally a sequence of decimal numbers separated by periods, such as '"1.18"'. The only reasonable use of this macro is to incorporate it into a string constant.

__OPTIMIZE__
This macro is defined in optimizing compilations. It causes certain GNU header files to define alternative macro definitions for some system library functions. It is unwise to refer to or test the definition of this macro unless you make very sure that programs will execute with the same effect regardless.

__CHAR_UNSIGNED__
This macro is defined if and only if the data type char is unsigned on the target machine. It exists to cause the standard header file 'limit.h' to work correctly.
It is bad practice to refer to this macro yourself; instead, refer to the standard macros defined in 'limit.h'.

3.4.3.2. Nonstandard Predefined Macros

The C preprocessor normally has several predefined macros that vary between machines because their purpose is to indicate what type of system and machine is in use. This manual, being for all systems and machines, cannot tell you exactly what their names are; instead, we offer a list of some typical ones.

Some nonstandard predefined macros describe the operating system in use, with more or less specificity. For example,

unix
'unix' is normally predefined on all Unix systems.

BSD
'BSD' is predefined on recent versions of Berkeley Unix (perhaps only in version 4.3).

Other nonstandard predefined macros describe the kind of CPU, with more or less specificity. For example,

vax
'vax' is predefined on Vax computers.

mc68000
'mc68000' is predefined on most computers whose CPU is a Motorola 68000, 68010 or 68020.

m68k
'm68k' is also predefined on most computers whose CPU is a 68000, 68010 or 68020; however, some makers use 'mc68000' and some use 'm68k'. Some predefine both names. What happens in GNU C depends on the system you are using it on.

M68020
'M68020' has been observed to be predefined on some systems that use 68020 CPUs---in addition to 'mc68000' and 'm68k' that are less specific.

-

ns32000
'ns32000' is predefined on computers which use the National Semiconductor 32000 series CPU.

Yet other nonstandard predefined macros describe the manufacturer of the system. For example,

sun
'sun' is predefined on all models of Sun computers.

pyr
'pyr' is predefined on all models of Pyramid computers.

sequent
'sequent' is predefined on all models of Sequent computers.

These predefined symbols are not only nonstandard, they are contrary to the ANSI standard because their names do not start with underscores. Therefore, the option '-ansi' inhibits the definition of these symbols.

This tends to make '-ansi' useless, since many programs depend on the customary nonstandard predefined symbols. Even system header files check them and will generate incorrect declarations if they do not find the names that are expected. You might think that the header files supplied for the Uglix computer would not need to test what machine they are running on, because they can simply assume it is the Uglix; but often they do, and they do so using the customary names. As a result, very few C programs will compile with '-ansi'. We intend to avoid such problems on the GNU system.

What, then, should you do in an ANSI C program to test the type of machine it is to run on?

GNU C offers a parallel series of symbols for this purpose, whose names are made from the customary ones by adding '__' at the beginning and end. Thus, the symbol __vax__ would be available on a vax, and so on.

The set of nonstandard predefined names in the GNU C preprocessor is controlled by the macro 'CPP_PREDEFINES', which should be a string containing '-D' options, separated by spaces. For example, on the Sun 3, we use the following definition:

```
#define CPP_PREDEFINES "-Dmc68000 -Dsun -Dunix -Dm68k"
```

3.4.4. Stringification

Stringification means turning a code fragment into a string constant whose contents are the text for the code fragment. For example, stringifying 'foo (z)' results in '"foo (z)"'.

In the C preprocessor, stringification is an option available when macro arguments are substituted into the macro definition. In the body of the definition, when an argument name appears, the character '#' before the name specifies stringification of the corresponding actual argument when it is substituted at that point in the definition. The same argument may be substituted in other places in the definition without stringification if the argument name appears in those places with no '#'.

Here is an example of a macro definition that uses stringification:

```
#define WARN_IF(EXP) \
  do { if (EXP) fprintf (stderr, "Warning: " #EXP "\n"); \
     } while (0)
```

Here the actual argument for 'EXP' is substituted once as given, into the 'if' statement, and once as stringified, into the argument to 'fprintf'. The 'do' and 'while (0)' are a kludge to make it possible to write 'WARN_IF (arg);', which the resemblance of 'WARN_IF' to a function would make C programmers want to do; see section Swallow Semicolon).

The stringification feature is limited to transforming one macro argument into one string constant: there is no way to combine the argument with other text and then stringify it all together. But the example above shows how an equivalent result can be obtained in ANSI Standard C using the feature that adjacent string constants are concatenated as one string constant. The preprocessor stringifies 'EXP''s actual argument into a separate string constant, resulting in text like

```
do  { if (x == 0) fprintf (stderr, "Warning: " "x == 0" "\n");
    } while (0)
```

but the C compiler then sees three consecutive string constants and concatenates them into one, producing effectively

```
do { if (x == 0) fprintf (stderr, "Warning: x == 0\n");
   } while (0)
```

Stringification in C involves more than putting double-quote characters around the fragment; it is necessary to put backslashes in front of all

around the fragment, it is necessary to put backslashes in front of all doublequote characters, and all backslashes in string and character constants, in order to get a valid C string constant with the proper contents. Thus, stringifying 'p = "foo\n";' results in '"p = \"foo\\n\";"'. However, backslashes that are not inside of string or character constants are not duplicated: '\n' by itself stringifies to '"\n"'.

Whitespace (including comments) in the text being stringified is handled according to precise rules. All leading and trailing whitespace is ignored. Any sequence of whitespace in the middle of the text is converted to a single space in the stringified result.

3.4.5. Concatenation

Concatenation means joining two strings into one. In the context of macro expansion, concatenation refers to joining two lexical units into one longer one. Specifically, an actual argument to the macro can be concatenated with another actual argument or with fixed text to produce a longer name. The longer name might be the name of a function, variable or type, or a C keyword; it might even be the name of another macro, in which case it will be expanded.

When you define a macro, you request concatenation with the special operator '##' in the macro body. When the macro is called, after actual arguments are substituted, all '##' operators are deleted, and so is any whitespace next to them (including whitespace that was part of an actual argument). The result is to concatenate the syntactic tokens on either side of the '##'.

Consider a C program that interprets named commands. There probably needs to be a table of commands, perhaps an array of structures declared as follows:

```
struct command

  char *name;
  void (*function) ();
};

struct command commands[] =
{
  {"quit", quit_command},
  {"help",help_command},
    ...
};
```

It would be cleaner not to have to give each command name twice, once in the string constant and once in the function name. A macro which takes the name of a command as an argument can make this unnecessary. The string constant can be created with stringification, and the function name by concatenating the argument with '_command'. Here is how it is done:

```
#define COMMAND(NAME) { #NAME, NAME ## _command }

struct command commands[] =
{
  COMMAND (quit),
  COMMAND (help),
    ...
};
```

The usual case of concatenation is concatenating two names (or a name and a number) into a longer name. But this isn't the only valid case. It is also possible to concatenate two numbers (or a number and a name, such as '1.5' and 'e3') into a number. Also, multi-character operators such as '+=' can be formed by concatenation. In some cases it is even possible to piece together a string constant. However, two pieces of text that don't together form a valid lexical unit cannot be concatenated. For example, concatenation with 'x' on one side and '+' on the other is not meaningful because those two characters can't fit together in any lexical unit of C. The ANSI standard says that such attempts at concatenation are undefined, but in the GNU C preprocessor it is well defined: it puts the 'x' and '+' side by side with no particular special results.

Keep in mind that the C preprocessor converts comments to whitespace before macros are even considered. Therefore, you cannot create a comment by concatenating '/' and '*': the '/*' sequence that starts a comment is not a lexical unit, but rather the beginning of a ''long'' space character. Also, you can freely use comments next to a '##' in a macro definition, or in actual arguments that will be concatenated, because the comments will be converted to spaces at first sight, and concatenation will later discard the spaces.

3.4.6. Undefining Macros

To undefine a macro means to cancel its definition. This is done with the

'#undef' command. '#undef' is followed by the macro name to be undefined.

Like definition, undefinition occurs at a specific point in the source file, and it applies starting from that point. The name ceases to be a macro name, and from that point on it is treated by the preprocessor as if it had never been a macro name.

For example,

```
#define FOO 4
x = FOO;
#undef FOO
x = FOO;
```

expands into

```
x = 4;
x = FOO;
```

In this example, 'FOO' had better be a variable or function as well as (temporarily) a macro, in order for the result of the expansion to be valid C code.

The same form of '#undef' command will cancel definitions with arguments or definitions that don't expect arguments. The '#undef' command has no effect when used on a name not currently defined as a macro.

3.4.7. Redefining Macros

Redefining a macro means defining (with '#define') a name that is already defined as a macro.

A redefinition is trivial if the new definition is transparently identical to the old one. You probably wouldn't deliberately write a trivial redefinition, but they can happen automatically when a header file is included more than once (see section Header Files), so they are accepted silently and without effect.

Nontrivial redefinition is considered likely to be an error, so it provokes a warning message from the preprocessor. However, sometimes it is useful to change the definition of a macro in mid-compilation. You can inhibit the warning by undefining the macro with '#undef' before the second definition.

In order for a redefinition to be trivial, the new definition must exactly match the one already in effect, with two possible exceptions:

- Whitespace may be added or deleted at the beginning or the end.
- Whitespace may be changed in the middle (but not inside strings). However, it may not be eliminated entirely, and it may not be added where there was no whitespace at all.

Recall that a comment counts as whitespace.

3.4.8. Pitfalls and Subtleties of Macros

In this section we describe some special rules that apply to macros and macro expansion, and point out certain cases in which the rules have counterintuitive consequences that you must watch out for.

3.4.8.1. Improperly Nested Constructs

Recall that when a macro is called with arguments, the arguments are substituted into the macro body and the result is checked, together with the rest of the input file, for more macro calls.

It is possible to piece together a macro call coming partially from the macro body and partially from the actual arguments. For example,

```
#define double(x) (2*(x))
#define call_with_1(x) x(1)
```

would expand 'call_with_1 (double)' into '(2*(1))'.

Macro definitions do not have to have balanced parentheses. By writing an unbalanced open parenthesis in a macro body, it is possible to create a macro call that begins inside the macro body but ends outside of it. For example,

```
#define strange(file) fprintf (file, "%s %d",
  ...
strange(stderr) p, 35)
```

This bizarre example expands to:

```
fprintf (stderr,  "%s  %d", p, 35)'!
```

3.4.8.2. Unintended Grouping of Arithmetic

You may have noticed that in most of the macro definition examples shown above, each occurrence of a macro argument name had parentheses around it.

In addition, another pair of parentheses usually surround the entire macro defin ition. Here is why it is best to write macros that way.

Suppose you define a macro as follows,

```
#define ceil_div(x, y) (x + y - 1) / y
```

whose purpose is to divide, rounding up. (One use for this operation is to compute how many 'int''s are needed to hold a certain number of 'char''s.)

Then suppose it is used as follows:

```
a = ceil_div (b & c, sizeof (int));
```

This expands into:

```
a = (b & c + sizeof (int) - 1) / sizeof (int);
```

which does not do what is intended. The operator-precedence rules of C make it equivalent to this:

```
a = (b & (c + sizeof (int) - 1)) / sizeof (int);
```

But what we want is this:

```
a = ((b & c) + sizeof (int) - 1)) / sizeof (int);
```

Defining the macro as:

```
#define ceil_div(x, y) ((x) + (y) - 1) / (y)
```

provides the desired result.

However, unintended grouping can result in another way. Consider 'sizeof ceil_div(1, 2)'. That has the appearance of a C expression that would compute the size of the type of 'ceil_div (1, 2)', but in fact it means something very different. Here is what it expands to:

```
sizeof ((1) + (2) - 1) / (2)
```

This would take the size of an integer and divide it by two. The precedence rules have put the division outside the 'sizeof' when it was intended to be inside.

Parentheses around the entire macro definition can prevent such problems. Here, then, is the recommended way to define 'ceil_div':

```
#define ceil_div(x, y) (((x) + (y) - 1) / (y))
```

3.4.8.3. Swallowing the Semicolon

Often it is desirable to define a macro that expands into a compound statement. Consider, for example, the following macro, that advances a pointer (the argument 'p' says where to find it) across whitespace characters:

```
#define SKIP_SPACES (p, limit)   \
 { register char *lim = (limit); \
   while (p != lim) {            \
     if (*p++ != ' ') {          \
       p--; break; }}}
```

Here Backslash-Newline is used to split the macro definition, which must be a single line, so that it resembles the way such C code would be laid out if not part of a macro definition.

A call to this macro might be 'SKIP_SPACES (p, lim)'. Strictly speaking, the call expands to a compound statement, which is a complete statement with no need for a semicolon to end it. But it looks like a function call. So it minimizes confusion if you can use it like a function call, writing a semicolon afterward, as in 'SKIP_SPACES (p, lim);'

But this can cause trouble before 'else' statements, because the semicolon is actually a null statement. Suppose you write

```
if (*p != 0)
  SKIP_SPACES (p, lim);
else ...
```

The presence of two statements---the compound statement and a null statement---in between the 'if' condition and the 'else' makes invalid C code.

The definition of the macro 'SKIP_SPACES' can be altered to solve this problem, using a 'do ... while' statement. Here is how:

```
#define SKIP_SPACES (p, limit)     \
do { register char *lim = (limit); \
     while (p != lim) {            \
       if (*p++ != ' ') {          \
         p--; break; }}}           \
            while (0)
```

Now 'SKIP_SPACES (p, lim);' expands into

Now "SKIP_SPACES (p, lim);" expands into

```
do {...} while (0);
```

which is one statement.

3.4.8.4. Duplication of Side Effects

Many C programs define a macro 'min', for ''minimum'', like this:

```
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
```

When you use this macro with an argument containing a side effect, as shown here,

```
next = min (x + y, foo (z));
```

it expands as follows:

```
next = ((x + y) < (foo (z)) ? (x + y) : (foo (z)));
```

where 'x + y' has been substituted for 'X' and 'foo (z)' for 'Y'.

The function 'foo' is used only once in the statement as it appears in the program, but the expression 'foo (z)' has been substituted twice into the macro expansion. As a result, 'foo' might be called two times when the statement is executed. If it has side effects or if it takes a long time to compute, the results might not be what you intended. We say that 'min' is an unsafe macro.

The best solution to this problem is to define 'min' in a way that computes the value of 'foo (z)' only once. The C language offers no standard way to do this, but it can be done with GNU C extensions as follows:

```
#define min(X, Y)                       \
({ typeof (X) __x = (X), __y = (Y);   \
(__x < __y) ? __x : __y; })
```

If you do not wish to use GNU C extensions, the only solution is to be careful when using the macro 'min'. For example, you can calculate the value of 'foo (z)', save it in a variable, and use that variable in 'min':

```
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
...
{
  int tem = foo (z);
  next = min (x + y, tem);
}
```

(where I assume that 'foo' returns type 'int').

3.4.8.5. Self-Referential Macros

A self-referential macro is one whose name appears in its definition. A special feature of ANSI Standard C is that the self-reference is not considered a macro call. It is passed into the preprocessor output unchanged.

Let's consider an example:

```
#define foo (4 + foo)
```

where 'foo' is also a variable in your program.

Following the ordinary rules, each reference to 'foo' will expand into '(4 + foo)'; then this will be rescanned and will expand into '(4 + (4 + foo))'; and so on until it causes a fatal error (memory full) in the preprocessor.

However, the special rule about self-reference cuts this process short after one step, at '(4 + foo)'. Therefore, this macro definition has the possibly useful effect of causing the program to add 4 to the value of 'foo' wherever 'foo' is referred to.

In most cases, it is a bad idea to take advantage of this feature. A person reading the program who sees that 'foo' is a variable will not expect that it is a macro as well. The reader will come across the identifier 'foo' in the program and think its value should be that of the variable 'foo', whereas in fact the value is four greater.

The special rule for self-reference applies also to indirect self-reference. This is the case where a macro x expands to use a macro 'y', and 'y''s expansion refers to the macro 'x'. The resulting reference to 'x' comes indirectly from the expansion of 'x', so it is a self-reference and is not further expanded. Thus, after

```
#define x (4 + y)
#define y (2 * x)
```

'x' would expand into '(4 + (2 * x))'. Clear?

But suppose 'y' is used elsewhere, not from the definition of 'x'. Then the use of 'x' in the expansion of 'y' is not a self-reference because 'x' is

use of 'x' in the expansion of 'y' is not a self-reference because 'x' is not ''in progress''. So it does expand. However, the expansion of 'x' contains a reference to 'y', and that is an indirect self-reference now because 'y' is ''in progress''. The result is that 'y' expands to '(2 * (4 + y))'.

It is not clear that this behaviour would ever be useful, but it is specified by the ANSI C standard, so you need to understand it.

3.4.8.6. Separate Expansion of Macro Arguments

We have explained that the expansion of a macro, including the substituted actual arguments, is scanned over again for macro calls to be expanded.

What really happens is more subtle: first each actual argument text is scanned separately for macro calls. Then the results of this are substituted into the macro body to produce the macro expansion, and the macro expansion is scanned again for macros to expand.

The result is that the actual arguments are scanned twice to expand macro calls in them.

Most of the time, this has no effect. If the actual argument contained any macro calls, they are expanded during the first scan. The result therefore contains no macro calls, so the second scan does not change it. If the actual argument were substituted as given, with no prescan, the single remaining scan would find the same macro calls and produce the same results.

You might expect the double scan to change the results when a self-referential macro is used in an actual argument of another macro (see section Self-Reference): the self-referential macro would be expanded once in the first scan, and a second time in the second scan. But this is not what happens. The self-references that do not expand in the first scan are marked so that they will not expand in the second scan either.

The prescan is not done when an argument is stringified or concatenated. Thus,

```
#define str(s) #s
#define foo 4
str (foo)
```

expands to '"foo"'. Once more, prescan has been prevented from having any noticeable effect.

More precisely, stringification and concatenation use the argument as written, in un-prescanned form. The same actual argument would be used in prescanned form if it is substituted elsewhere without stringification or concatenation.

```
#define str(s) #s lose(s)
#define foo 4
str (foo)
```

expands to '"foo" lose(4)'.

You might now ask, ''Why mention the prescan, if it makes no difference? And why not skip it and make the preprocessor faster?'' The answer is that the prescan does make a difference in three special cases:

- Nested calls to a macro.
- Macros that call other macros that stringify or concatenate.
- Macros whose expansions contain unshielded commas.

We say that nested calls to a macro occur when a macro's actual argument contains a call to that very macro. For example, if 'f' is a macro that expects one argument, 'f (f (1))' is a nested pair of calls to 'f'. The desired expansion is made by expanding 'f (1)' and substituting that into the definition of 'f'. The prescan causes the expected result to happen. Without the prescan, 'f (1)' itself would be substituted as an actual argument, and the inner use of 'f' would appear during the main scan as an indirect self-reference and would not be expanded. Here, the prescan cancels an undesirable side effect (in the medical, not computational, sense of the term) of the special rule for self-referential macros.

But prescan causes trouble in certain other cases of nested macro calls. Here is an example:

```
#define foo  a,b
#define bar(x) lose(x)
#define lose(x) (1 + (x))
bar(foo)
```

We would like 'bar(foo)' to turn into '(1 + (foo))', which would then turn into '(1 + (a,b))'. But instead, 'bar(foo)' expands into 'lose(a,b)', and

you get an error because lose requires a single argument. In this case, the problem is easily solved by the same parentheses that ought to be used to prevent misnesting of arithmetic operations:

```
#define foo (a,b)
#define bar(x) lose((x))
```

The problem is more serious when the operands of the macro are not expressions; for example, when they are statements. Then parentheses are unacceptable because they would make for invalid C code:

```
#define foo { int a, b; ... }
```

In GNU C you can shield the commas using the '({...})' construct which turns a compound statement into an expression:

```
#define foo ({ int a, b; ... })
```

Or you can rewrite the macro definition to avoid such commas:

```
#define foo { int a; int b; ... }
```

There is also one case where prescan is useful. It is possible to use prescan to expand an argument and then stringify it---if you use two levels of macros. Let's add a new macro 'xstr' to the example shown above:

```
#define xstr(s) str(s)
#define str(s) #s
#define foo 4
xstr (foo)
```

This expands into '"4"', not '"foo"'. The reason for the difference is that the argument of 'xstr' is expanded at prescan (because 'xstr' does not specify stringification or concatenation of the argument). The result of prescan then forms the actual argument for 'str'. 'str' uses its argument without prescan because it performs stringification; but it cannot prevent or undo the prescanning already done by 'xstr'.

3.4.8.7. Cascaded Use of Macros

A cascade of macros is when one macro's body contains a reference to another macro. This is very common practice. For example,

```
#define BUFSIZE 1020
#define TABLESIZE BUFSIZE
```

This is not at all the same as defining 'TABLESIZE' to be '1020'. The '#define' for 'TABLESIZE' uses exactly the body you specify---in this case, 'BUFSIZE'---and does not check to see whether it too is the name of a macro.

It's only when you use 'TABLESIZE' that the result of its expansion is checked for more macro names.

This makes a difference if you change the definition of 'BUFSIZE' at some point in the source file. 'TABLESIZE', defined as shown, will always expand using the definition of 'BUFSIZE' that is currently in effect:

```
#define BUFSIZE 1020
#define TABLESIZE BUFSIZE
#undef BUFSIZE
#define BUFSIZE 37
```

Now 'TABLESIZE' expands (in two stages) to '37'.

3.5. Conditionals

In a macro processor, a conditional is a command that allows a part of the program to be ignored during compilation, on some conditions. In the C preprocessor, a conditional can test either an arithmetic expression or whether a name is defined as a macro.

A conditional in the C preprocessor resembles in some ways an 'if' statement in C, but it is important to understand the difference between them. The condition in an 'if' statement is tested during the execution of your program. Its purpose is to allow your program to behave differently from run to run, depending on the data it is operating on. The condition in a preprocessor conditional command is tested when your program is compiled. Its purpose is to allow different code to be included in the program depending on the situation at the time of compilation.

3.5.1. Why Conditionals are Used

Generally there are three kinds of reason to use a conditional.

- A program may need to use different code depending on the machine or operating system it is to run on. In some cases the code for one operating system may be erroneous on another operating system; for example, it might refer to library routines that do not exist on the other system. When this happens, it is not enough to avoid executing

other system. When this happens, it is not enough to avoid executing the invalid code: merely having it in the program makes it impossible to link the program and run it. With a preprocessor conditional, the offending code can be effectively excised from the program when it is not valid.

- You may want to be able to compile the same source file into two different programs. Sometimes the difference between the programs is that one makes frequent time-consuming consistency checks on its intermediate data while the other does not.
- A conditional whose condition is always false is a good way to exclude code from the program but keep it as a sort of comment for future reference.

Most simple programs that are intended to run on only one machine will not need to use preprocessor conditionals.

3.5.2. Syntax of Conditionals

A conditional in the C preprocessor begins with a conditional command : '#if', '#ifdef' or '#ifndef'. See section Conditionals-Macros, for info on '#ifdef' and '#ifndef'; only '#if' is explained here.

3.5.2.1. The '#if' Command

The '#if' command in its simplest form consists of

```
#if expression
controlled text
#endif /* expression */
```

The comment following the '#endif' is not required, but it is a good practice because it helps people match the '#endif' to the corresponding '#if'. Such comments should always be used, except in short conditionals that are not nested. In fact, you can put anything at all after the '#endif' and it will be ignored by the GNU C preprocessor, but only comments are acceptable in ANSI Standard C.

expression is a C expression of integer type, subject to stringent restrictions. It may contain

- Integer constants, which are all regarded as long or unsigned long.
- Character constants, which are interpreted according to the character set and conventions of the machine and operating system on which the preprocessor is running. The GNU C preprocessor uses the C data type 'char' for these character constants; therefore, whether some character codes are negative is determined by the C compiler used to compile the preprocessor. If it treats 'char' as signed, then character codes large enough to set the sign bit will be considered negative; otherwise, no character code is considered negative.
- Arithmetic operators for addition, subtraction, multiplication, division, bitwise operations, shifts, comparisons, and '&&' and '||'.
- Identifiers that are not macros, which are all treated as zero(!).
- Macro calls. All macro calls in the expression are expanded before actual computation of the expression's value begins.

Note that 'sizeof' operators and enum-type values are not allowed. enum-type values, like all other identifiers that are not taken as macro calls and expanded, are treated as zero.

The text inside of a conditional can include preprocessor commands. Then the commands inside the conditional are obeyed only if that branch of the conditional succeeds. The text can also contain other conditional groups. However, the '#if''s and '#endif''s must balance.

3.5.2.2. The '#else' Command

The '#else' command can be added to a conditional to provide alternative text to be used if the condition is false. This looks like

```
#if expression
text-if-true
#else /* Not expression */
text-if-false
#endif /* Not expression */
```

If expression is nonzero, and the text-if-true is considered included, then '#else' acts like a failing conditional and the text-if-false is ignored. Contrariwise, if the '#if' conditional fails, the text-if-false is considered included.

3.5.2.3. The '#elif' Command

One common case of nested conditionals is used to check for more than two possible alternatives. For example, you might have

```
#if X == 1
...
#else /* X != 1 */
#if X == 2
...
#else /* X != 2 */
...
#endif /* X != 2 */
#endif /* X != 1 */
```

Another conditional command, '#elif', allows this to be abbreviated as follows:

```
#if X == 1
...
#elif X == 2
...
#else /* X != 2 and X != 1*/
...
#endif /* X != 2 and X != 1*/
```

'#elif' stands for ''else if''. Like '#else', it goes in the middle of a '#if'-'#endif' pair and subdivides it; it does not require a matching '#endif' of its own. Like '#if', the '#elif' command includes an expression to be tested.

The text following the '#elif' is processed only if the original '#if'-condition failed and the '#elif' condition succeeeds. More than one '#elif' can go in the same '#if'-'#endif' group. Then the text after each '#elif' is processed only if the '#elif' condition succeeds after the original '#if' and any previous '#elif''s within it have failed. '#else' is equivalent to '#elif 1', and '#else' is allowed after any number of '#elif''s, but '#elif' may not follow a '#else'.

3.5.3. Keeping Deleted Code for Future Reference

If you replace or delete a part of the program but want to keep the old code around as a comment for future reference, the easy way to do this is to put '#if 0' before it and '#endif' after it.

This works even if the code being turned off contains conditionals, but they must be entire conditionals (balanced '#if' and '#endif').

3.5.4. Conditionals and Macros

Conditionals are rarely useful except in connection with macros. A '#if' command whose expression uses no macros is equivalent to '#if 1' or '#if 0'; you might as well determine which one, by computing the value of the expression yourself, and then simplify the program. But when the expression uses macros, its value can vary from compilation to compilation.

For example, here is a conditional that tests the expression 'BUFSIZE == 1020', where 'BUFSIZE' must be a macro.

```
#if BUFSIZE == 1020
  printf ("Large buffers!\n");
#endif /* BUFSIZE is large */
```

The special operator 'defined' may be used in '#if' expressions to test whether a certain name is defined as a macro. Either 'defined name ' or 'defined (name)' is an expression whose value is 1 if name is defined as macro at the current point in the program, and 0 otherwise. For the 'defined' operator it makes no difference what the defini-tion of the macro is; all that matters is whether there is a definition. Thus, for example,

```
#if defined (vax) || defined (ns16000)
```

would include the following code if either of the names 'vax' and 'ns16000' is defined as a macro.

If a macro is defined and later undefined with '#undef', subsequent use of the 'defined' operator will return 0, because the name is no longer defined. If the macro is defined again with another '#define', 'defined' will recommence returning 1.

Conditionals that test just the definedness of one name are very common, so there are two special short conditional commands for this case. They are

#ifdef name

is equivalent to '#if defined (name)'.

#ifndef name

is equivalent to '#if ! defined (name)'.

Macro definitions can vary between compilations for several reasons.

- Some macros are predefined on each kind of machine. For example, on a Vax, the name 'vax' is a predefined macro. On other machines, it would not be defined.
- Many more macros are defined by system header files. Different systems and machines define different macros, or give them different values. It is useful to test these macros with conditionals to avoid using a system feature on a machine where it is not implemented.
- Macros are a common way of allowing users to customize a program for different machines or applications. For example, the macro 'BUFSIZE' might be defined in a configuration file for your program that is included as a header file in each source file. You would use 'BUFSIZE' in a preprocessor conditional in order to generate different code depending on the chosen configuration.
- Macros can be defined or undefined with '-D' and '-U' command options when you compile the program. You can arrange to compile the same source file into two different programs by choosing a macro name to specify which program you want, writing conditionals to test whether or how this macro is defined, and then controlling the state of the macro with compiler command options. See section Invocation.

3.5.5. The '#error' Command

The command '#error' causes the preprocessor to report a fatal error. The rest of the line that follows '#error' is used as the error message.

You would use '#error' inside of a conditional that detects a combination of parameters which you know the program does not properly support. For example, if you know that the program will not run properly on a Vax, you might write

```
#ifdef vax
#error Won't work on Vax. See comments at get_last_obj.
#endif
```

See section Nonstandard Predefined, for why this works.

If you have several configuration parameters that must be set up by the installation in a consistent way, you can use conditionals to detect an inconsistency and report it with '#error'. For example,

```
#if HASH_TABLE_SIZE % 2 == 0 || HASH_TABLE_SIZE % 3 == 0 \
                             || HASH_TABLE_SIZE % 5 == 0
#error HASH_TABLE_SIZE not be divisible by a small prime
#endif
```

3.6. Combining Source Files

One of the jobs of the C preprocessor is to inform the C compiler of where each line of C code came from: which source file and which line number.

C code can come from multiple source files if you use '#include'; both '#include' and the use of conditionals and macros can cause the line number of a line in the preprocessor output to be different from the line's number in the original source file. You will appreciate the value of making both the C compiler (in error messages) and symbolic debuggers such as GDB use the line numbers in your source file.

The C preprocessor builds on this feature by offering a command by which you can control the feature explicitly. This is useful when a file for input to the C preprocessor is the output from another program such as the bison parser generator, which operates on another file that is the true source file. Parts of the output from bison are generated from scratch, other parts come from a standard parser file. The rest are copied nearly verbatim from the source file, but their line numbers in the bison output are not the same as their original line numbers. Naturally you would like compiler error messages and symbolic debuggers to know the original source file and line number of each line in the bison output.

bison arranges this by writing '#line' commands into the output file. '#line' is a command that specifies the original line number and source file name for subsequent input in the current preprocessor input file. '#line' has three variants:

#line linenum

Here linenum is a decimal integer constant. This specifies that the

line number of the following line of input, in its original source file, was linenum .

#line linenum filename

Here linenum is a decimal integer constant and filename is a string constant. This specifies that the following line of input came originally from source file filename and its line number there was linenum . Keep in mind that filename is not just a file name; it is surrounded by doublequote characters so that it looks like a string constant.

#line anything else

anything else is checked for macro calls, which are expanded. The result should be a decimal integer constant followed optionally by a string constant, as described above.

'#line' commands alter the results of the '__FILE__' and '__LINE__' predefined macros from that point on. See section Standard Predefined.

3.7. Miscellaneous Preprocessor Commands

This section describes three additional preprocessor commands. They are not very useful, but are mentioned for completeness.

The null command consists of a '#' followed by a Newline, with only whitespace (including comments) in between. A null command is understood as a preprocessor command but has no effect on the preprocessor output. The primary significance of the existence of the null command is that an input line consisting of just a '#' will produce no output, rather than a line of output containing just a '#'. Supposedly some old C programs contain such lines.

The ' **#pragma** ' command is specified in the ANSI standard to have an arbitrary implementation-defined effect. In the GNU C preprocessor, '#pragma' commands are ignored, except for '#pragma once' (see section Once-Only).

The ' **#ident** ' command is supported for compatibility with certain other systems. It is followed by a line of text. On certain systems, the text is copied into a special place in the object file; on most systems, the text is ignored and this directive has no effect.

3.8. C Preprocessor Output

The output from the C preprocessor looks much like the input, except that all preprocessor command lines have been replaced with blank lines and all comments with spaces. Whitespace within a line is not altered; however, a space is inserted after the expansions of most macro calls.

Source file name and line number information is conveyed by lines of the form

filename linenum flag

which are inserted as needed into the middle of the input (but never within a string or character constant). Such a line means that the following line originated in file filename at line linenum .

The third field, flag , may be a number, or may be absent. It is '1' for the beginning of a new source file, and '2' for return to an old source file at the end of an included file. It is absent otherwise.

4. Acknowledgements.

GNU cpp was ported to QDOS by Dave Woodman.

C68 Compiler

NAME

c68/c386/c86/c30 - Compile preprocessed C source.

SYNOPSIS

c68 [options] [input_file [output_file [listing_file]]]

c386 [options] [input_file [output_file [listing_file]]]

c86 [options] [input_file [output_file [listing_file]]]

c30 [options] [input_file [output_file [listing_file]]]

cARM [options] [input_file [output_file [listing_file]]]

cPPC [options] [input_file [output_file [listing_file]]]

DESCRIPTION

c68 is a publicly available ANSI C compiler. The compiler can also operate in a mode that is compatible with the original Kernighan and Richie (K&R) definition. The user can select K&R mode (which causes many ANSI specific features to be disabled) by a run-time parameter option. The default is ANSI compatible mode as this is what most people would wish to use.

Although generically the compiler is known under the name of c68 the actual program name normally varies according to the target environment in which it is hosted. The names used are c68 when hosted on Motorola 680x0 systems; c386 when hosted on Intel 386 (or better) systems running in 32 bit mode; c86 when hosted on an Intel 8086 (or better system) running in 16-bit real mode; cARM when hosted on a Acorn ARM Risc processor; c30 when hosted on a Texas Instruments TMSC30 processor; cPPC when hosted on a Power PC processor.

The compiler was originally developed to run on the MINIX operating system, but is known to be in wide use on a wide variety of other operating systems such as LINUX, TOS (Atari), QDOS/SMSQ (Sinclair QL) and EPOC (Psion 3a). The source has been specifically written with maximum portability in mind.

The compiler is slightly unusual in there is the capability for simultaneous support of multiple target processor types and/or multiple assemblers. The user specifies at the time that the compiler is built which target processors and combination of options are to be supported. If support for multiple processors and/or assemblers are configured to be built in, then options other than the default can be selected by appropriate runtime parameter options. This can make the compiler very useful as a tool for cross-development between different hardware platforms.

The compiler takes the output of a C pre-processor, and compiles it to give assembler source. If no output file is specified on the command line then the compiler writes the generated assembler code to standard output. If in addition there is no input file specified then the compiler reads the C source from standard input. Finally if the compiler run time option requesting a listing is used and no listing file is specified, the compiler writes it to standard error.

The options available to control the behaviour of the compiler are listed below. The options to the compiler can also be passed as -Qoption in addition to the syntax given below. This is to make it easier for the front-end programs (typically called CC) to decide which options belong to the compiler phase. Not all options are necessarily available in all versions of the compiler as some of them are dependent on the settings in the compiler configuration file at the time that the compiler is actually built.

GENERAL OPTIONS

These are options that are not dependent on the target processor type, and are general in nature. Where the option can be a list, then multiple options from the list can be specified separated by commas. There must be no spaces between the options in this case.

-?

Display a message giving the full list of options available in this particular version of the compiler. It also details the default settings for the parameters. This option can be very useful as it always reflects the choices of settings in the compiler configuration file that were actually used when generating this version of the compiler. If you find a parameter option listed in this document appears to be ignored, then this is the way to check if the version of the compiler you are actually using has been built with that option enabled.

The values listed are organised so that the first section applies to setting global to all variants of the compiler, and then sections specific to each target processor type for which support has been included when the compiler was built.

The output is normally too long to fit onto a single screen, so you may need to redirect into a file to see all the options.

-v

Output additional information during the compile process. If the compiler was built without the **VERBOSE** configuration option set then this is merely a message giving the version number of the compiler. If the **VERBOSE** configuration option was used when the compiler was built, then additional progress information is output during the compile process.

Default: The compiler as supplied is not normally built with the **VERBOSE** option and merely provides the version number message if **-v** is used.

-warn= n
Control the severity level of warning and diagnostic messages that will be output during the compilation process. Messages with a higher severity value (i.e. less severe) than the value specified will not be output. See later for more information on the effect of possible values for **n** .

Default: **-warn=3**

-error= n
Make messages that are normally only warnings to be treated as errors instead. The value of **n** specifies what severity of messages that would normally be only warnings are instead to be treated as errors. This option is often used in conjunction with the **maxerr** option..

Default: **-error=0**

-maxerr= n
Sets the maximum error count to the value of **n** . This is the maximum number of errors that will be reported before the compiler abandons a compile. As one error can cause others to occur, in a cascade effect, it is often a good idea to set this to a low value in the region of 10-20 errors which fits on one screen.

Default: **-maxerr=200**

-debug= option_list
This option is only available in a version of the compiler built with the **DEBUG** configuration option defined. It is used to control the amount of debug information that is written to the listing file. The option_list can be any combination of the following:

global
peep
expr
code
register
symbol

Default: No debug information output.

N.B. The **DEBUG** configuration option when building the compiler is normally only set if you are developing new code to be included in the compiler or investigating faults. It is therefore never normally included in any generally distributed binaries.

-align= yes **|** no
All processors tend to have default alignments at which they generate most efficient code. The compiler will use the setting of the **-align** option to decide whether to use the processor optimum alignment, or ask the compiler to attempt to use a different alignment. A **yes** value for this option means align structures and unions using the same rules as applied to the member that has the strictest alignment rules, while **no** means use the default value for the processor type. In particular if you want structures or unions which only contain 'char' data types to be packed as closely as possible (and therefore possibly start on odd addresses) you must use the **-align=yes** setting.

Default:
-align=yes for Intel targets (any boundaries)
-align=no for 68000 and ARM targets (even boundaries)
-align=yes for TMS C30 targets (even boundaries)

-asm= yes **|** no
Specifies whether the use of the **asm** keyword should be allowed in your C source. Use of the **asm** keyword is not part of ANSI C and will definitely result in non-portable code.

Support for the **asm** keyword is not included in the compiler unless the **ASM** configuration option is set at the time the compiler is built. We do not normally include such support in binaries we put on general distribution.

Default: **-asm=no**

-extension= yes **|** no
Specifies whether options that are under consideration for inclusion in the next ANSI C standard (amendment 2) should be included.

Support for these options will only be included in the compiler if the **EXTENSIONS** configuration option is set when the compiler is built. We normally do include such an option in binaries we put on general distribution except when the size of the compiler is constrained by memory limits. For details of what options are affected by this keyword refer to the section later in this document labelled "EXTENSIONS TO ANSI C".

Default: **-extensions=no**

-extern= yes **|** no
Output details of external symbols in this module to the listing file. This is intended in the future to provide the basis of a lint-style facility to provide cross-module consistency checking.

Whether support for this option is included in the compiler is controlled by the **EXTERNAL** configuration option at the time the compiler is built. We do not normally include such support in binaries we put on general distribution.

Default: **-extern=no**

-fcheck= yes **|** no
This option is only relevant in versions of the compiler that were configured at build time to not include support for floating point, but that did have the **FLOAT_CHECK** configuration option set. Setting **-fcheck=yes** means that floating point keywords will be recognised and you will get errors output if you try and use such keywords. Setting **-fcheck=no** means that these keywords are not recognised as C keywords.

Default: **fcheck=no**

-format= yes **|** no
Activate additional checks for the 'printf' and 'scanf' families of library routines. If active, then the parameters following the format string are checked as being compatible with the format string.

This option is only available if the **FORMAT_CHECK** configuration option was set at the time the compiler it is built. This option is very useful so we normally try and include it, but the support is sometimes removed to save memory when this is critical.

Default: **-format=yes** (check parameters)

-icode
Output run-time debugging information to the listing file. Intended mainly for debugging the compiler itself.

This option is only available if the compiler was built with the **ICODE** configuration option defined. This option is not normally defined for binaries that we put on general distribution.

Default: **-icode=no**

-int= 16 **|** 32
Specify whether the length of **int** declarations should be 16 bit (same as a short) or a 32 bit (same as a long). There is a lot of code around that assumes **sizeof(int)==sizeof(char *)** so getting this setting correct for your target platform is important.

Default:
c386: **-int=32**
c68: **-int=16** (MINIX systems)
-int=32 (QDOS/SMS systems)
c86: **-int=16** (Psion 3a systems)
c30: **-int=32**

-lattice= yes **|** no
Older versions of Lattice C had partial support of prototypes in which a variable number of parameters was indicated by finishing the parameter list with a comma (rather than the ANSI style of using ,...). The use of this option means the Lattice syntax will also be accepted.

Default: **-lattice=no**

-list= yes **|** no
Control listing of symbol table.

Support for this option is only available if the **LIST** configuration

Support for this option is only available if the **DEBUG** configuration option was included when the compiler was built. This option is primarily an aid to helping us debug the compiler, so support for this option would not normally be included in any distribute binaries.

Default: **-list=no**

-obsolete= yes **|** no
Specifies whether warnings should be generated if you use an option that is currently part of the ANSI C standard, but which the ANSI committee have warned may be removed from future versions of the ANSI C standard. Examples of this is support for K&R style function definitions.

Default: **-obsolete=no** (no warnings)

-packenum= yes **|** no
Specify whether the compiler should use the smallest integer type that is capable of containing all the enumeration values that are defined for a particular enumeration type. If **-packenum=no** is in effect then ' **int** ' is used as the enumeration type.

This option is only supported if the **PACKENUM** configuration option was set at the time the compiler was built. We normally do have this option supported in any binaries we put on general distribution.

Default: **-packenum=no**

-revbit= yes **|** no
Control the order in which the compiler allocates the bits in a bitfield. The **-revbit=yes** option causes the bitfield to be allocated starting from the highest number bit downwards, rather than the default of allocating them from bit 0 upwards.

Default: **-revbit=no** (start at bit 0)

-topspeed= yes **|** no
Control whether certain specific extensions to the C syntax that are used by the TopSpeed C compiler should be treated as valid or not.

N.B. The fact that the syntax is accepted does not mean that the same effect will be obtained as when used under TopSpeed - in most cases the additional information is simply ignored.

Whether this option is supported is determined by whether the **TOPSPEED** configuration option was set at the time the compiler was built.

Default: **-topspeed=no**

-trad= yes **|** no
Determine whether the compiler should reject most of the ANSI extensions to the original K&R definition and work instead in "traditional" mode. For more detail on what ANSI options are not supported when this option is set, see the section later in this document on K&R Compatibility Mode.

Default: **-trad=no**

-uchar=yes
Specifies whether the **char** data type is considered as an unsigned integer type with values in the range 0 to 255, or a signed integer type with the range +127 to -128.

Default: **-uchar=no** (signed char)

GENERAL CODE GENERATION OPTIONS

These are options that affect the code generation process, but that are not dependent on the target processor type.

-g
Output additional information for debugging purposes. Branch optimisation is also suppressed even if the **-O** option has been specified. The current effect of this option is to include **line** directives in the generated assembler output, plus the text of the current source line as a comment. Not all the assembler can accept the **line** directive, so you may find that you cannot generate the object code from such an assembler source file. This can still be useful if you wish to see exactly which C source lines caused particular assembler code to be generated.

Default: No debugging information is generated.

-O
Specifies that maximum optimisation available is to be used. This can significantly reduce the size of the generated code, and will also normally slightly improve on run time. It can, however, slow down the compilation process. You can also use the **-peep** option to turn on just certain parts of the optimisation process.

Note that this option is ignored if the **-g** or **-opt=no** options are also specified in the command line.

Default: The optimisation triggered by this option is not performed.

-code= yes **|** no
Specifies whether code is to be generated, or if this run is merely being used to check for errors in the source code. The advantage of specifying the **-code=no** option if you are merely looking for errors is that the compiler will run faster if no attempt is made to generate code.

Default: **-code=yes**

-longdouble= yes **|** no
If set to 'yes' then 'long double' is treated as being a distinct type from 'double' with different support routines.

Default: **-longdouble=no**

NOTE. The software support routines for 'long double' are not currently available for use with **c386** / **c86** / **c68** so you would normally only consider using this option if generating inline FPU instructions.

-opt= yes **|** no
Control the operation of the global optimiser. Normally the optimiser is active as it results in more efficient code. If you wish to suppress all global optimisations then you can specify the **-opt=no** option. You would not normally use this option unless you suspect an error in the optimiser. Using the **-opt=no** option will override the **-O** option if it is also specified.

Default: **-opt=yes**

-prefix= string
This allows the prefix that is added to external symbol names (normally either an underscore character, or a null string) to be changed. The compiler takes whatever follows the equals sign as the string value. Quotes should NOT be added unless required by the parameter parsing mechanism of the host operating system.

Default: This is really determined by the standards of the target operating system. As issued the setting is:

-prefix=_

-reg= yes **|** no
Specifies constraints on how the compiler is allowed to allocate variables to registers. Normally the compiler will try to do automatic allocation of variables to registers according to their run-time usage. The **-reg=no** option forces the use of register variables only when explicitly requested by the programmer.

Default: **-reg=yes**

-separate= yes **|** no
Determine whether the compiler should allocate strings and constants in the same segment as the code, or in a separate data segment.

Default: **-separate=no**

-stackcheck= yes **|** no
Specify whether calls should be made to a support routine to perform stack checks at the start of each function. To use this option, it is necessary to have implemented the appropriate (system dependent) support routine.

Default: - **stackcheck=no**

-stackopt= safest **|** minimum **|** average **|** maximum
Used to control whether the 'lazy stack updating' optimisation is to be

used. The meanings of the various values are:

safest Suppress this level of optimisation. It is advisable to suppress lazy stack optimisation on routines which are recursive in nature. Failure to do so may lead to excessive stack space being required to successfully run this program.

minimum A certain amount of optimisation is done, but nothing that is considered dangerous. This is the safest mode of optimisation assuming you allow this type of optimisation at all.

average Allow optimisation for functions whose name starts with an underscore.

maximum Allow optimisation for functions whose name starts with an underscore, or which are called via a function variable. This effectively optimises all function calls.

See the section on optimisation later in this document for more detail on the implications of the various settings for this optimisation.

Default: **-stackopt=minimum**

-trace= yes **|** no
Control the generation of run-time trace information. Intended in the future to help support a source code debugger. However, at the moment this capability is incomplete. This option is only available if the compiler was built with the **TRACE** configuration option set. The compiler as normally supplied is not set to have this option built in.

Default: **-trace=no**

-trans= yes **|** no
This option is used if you are working on a system which can only support symbol names of limited length. It allows you to make certain that all names in the assembler output are only 8 characters in length (a special algorithm is used for names that are longer than this). This is used if the assembler phase cannot handle long C names. Support for this option is only included if the **TRANSLATE** configuration option was set when the compiler was built. As most modern systems can support longer symbol names we normally omit support for this option in binaries that are put on general distribution.

Default: **-trans=no**

MOTOROLA 68K OPTIONS

The options listed in this section apply when generating code for Motorola 68K family of processors. They will only be available if support for the Motorola 68K processors was specified at the time the compiler was built.

-codemode= absolute **|** small **|** large
This option is used to tell the compiler what addressing modes to use for jump instructions. The meanings of the options are:

absolute Any generated jump instructions use absolute addressing mode. Typically this means that runtime relocation of the generated program needs to be done.

small Jump instructions will use relative addressing modes assuming the 'small model'. This means that all target of jump instructions are within a 16-bit displacement of the source. If you set this and it turns out not to be true you will almost certainly get errors when you try and link your code.

large Jump instructions will use relative addressing modes assuming the 'large model' This means that all targets of jump instructions are within a 32-bit displacement of the source (which will always be true).

Note that although the **small** and **large** options generate position independent code, the resulting program will not be position independent unless any supplied libraries you intend to use have also been generated to use this option, and any variables (that are not auto variables) are accessed using position relative addressing.

Default: **-codemodel=absolute**

-datamodel= absolute **|** small **|** large

This option is used to tell the compiler what addressing modes to use for accessing program variables. The meanings of the options are:

absolute Variables are accessed using absolute mode. Typically this means that runtime relocation of the generated program needs to be done.

small Variables are accessed using 'small model' addressing. This means 16-bit displacements from the register specified in the **-regdata** parameter. If you set this and it turns out not to be true you will almost certainly get errors when you try and link your code.

large Variables are accessed using 'large model' addressing. This means 32-bit displacements from the register specified in the **-regdata** parameter.

Note that the code generated by the **small** and **large** options requires the address of a specific external label to be loaded into the register specified by the **-regdata** parameter as all addresses are generated as displacements from this label. Typically this is done in the program start-up module.

Only change this option from the default if your library supplier tells you that you can, or you are **very** sure you know what you are doing. Also there is normally not much to be gained from using this option unless any supplied libraries you intend to use have also been generated to use this option.

Default: **-datamodel=absolute**

-fpu= yes **|** no
Specify whether operations involving floating point variables should generate in-line calls to a hardware floating point unit, or whether calls are made instead to library support routines. Using library support routines allows floating point operations to be carried out purely in software.

Default: **-fpu=no**

-fpureturn =yes **|** no
This option is used to tell the compiler whether the library routines are such that floating point results are returned in the hardware FPU registers, or in normal registers. Note you should not normally change this value from the default unless you have been specifically advised to do so.

Default: **-fpureturn=no**

-interrupt =yes **|** no
This option is used to decide whether functions should be terminated with a RTS or a RTE instruction. You would want a RTS in normal code, and a RTE in an interrupt handler.

The way that you would most likely use this option is by by using it with an inline #pragma statement rather than as a command line option. In other words along the lines of:

```
#pragma interrupt=yes
void special_func()
{
.... code for function
}
#pragma interrupt=no
```

This would have the effect of only the single function specified having a RTE instruction to terminate it with all others having an RTS as the return instruction.

Default: **-interrupt=no**

-peep= none **|** peepopt_list **|** all
Control the level of peephole optimisation that should be done. Past experience has shown that some of the more obscure bugs reported on the compiler are those where the peephole optimiser part of the compiler has made an invalid optimisation. You would therefore use this option if you suspect that the compiler has generated incorrect code, and you want to look at what would be generated if some or all of the the peephole optimisation was not done.

The meanings of the options are:

none All peephole optimisations are suppressed.

all All peephole optimisations are performed.

You can also exercise a finer level of control by specifying the exact combination of peephole optimisations that you want from the following options:

instruction This controls whether instruction sequences should be replaced by more efficient combinations.

jumps This controls whether jump optimisation should be used which tries to common up re-occurring bits of code. This normally produces significant size savings in the generated code.

flow This tries to analyse the flow of the code to eliminate redundant loads of registers. A significant size savig normally results from this option. However if it goes wrong, the results can be rather unpredictable.

Default: **-peep=all**

-probe= no **|** yes
Specify whether stack probes should be generated each time a stack frame is generated. These can be desirable if using the compiler in a multi-tasking environment and with a 680x0 based system which has hardware protection for invalid memory accesses. The problem is that not enough space is always left on the stack to store information for a restart of an instruction, and stack probes insure that the stack has enough allocated memory to accommodate the needs of the routine. Support for this option is only included if the **PROBES** configuration option was set when the compiler was built.

Default: **-probe=no**

Note: The compiler as supplied is not normally set to have this option compiled in. Also there is no point in attempting to use it if your system does not have hardware that will detect attempts to access memory addresses that are outside the stack.

-regdata=a n
Specify which address register is to be used as the index register for access to variables. This parameter is only relevant if the setting of the **-datamode** specifies that absolute addressing is not being used. Note that no check is made that the settings do not conflict with any of the other **-regxxxx** options.

Default: **-regdata=a5**

-regframe=a n
Specify which address register is to be used as the frame pointer. Note that no check is made that the settings do not conflict with any of the other **-regxxxx** options.

Default: **-regframe=a6**

-regtemp= register_list
Specify which registers are treated as scratch registers. Note that no check is made to ensure that you have left enough for the compiler to be able to sensibly generate code, or that the settings do not conflict with any of the other **-regxxxx** options.

Default: **-regtemp=d0,d1,d2,a0,a1,fp0,fp1,fp2**

-regunused= register_list
Specify which registers should not be used. This would be used if you needed to ensure that a particular register was never corrupted for some reason. Note that issued libraries will not have been built with this setting, so use in average programs is not much use unless the libraries are rebuilt to match. Note that no check is made that the settings do not conflict with any of the other **-regxxxx** options.

Default: **-regunused=**

-target=n
Used to specify the target processor type. Values supported are:

68000
68010
68020
68030
68040

Default: **-target=68000**

If support for multiple processors and/or assemblers was configured when the compiler was built, then you can specify a 68k target with a specific assembler using the following options:

-ack68k
Generate 680x0 code. Use the ACK assembler syntax for the output.

-cpm68k
Generate 680x0 code. Use the CPM assembler syntax for the output.

-gas68K
Generate 680x0 code. Use the GNU assembler syntax for the output.

-qmc68k
Generate 680x0 code. Use the QMAC assembler syntax for the output.

INTEL 386 OPTIONS

The options in this section apply when generating 32-bit code for Intel 386 (or better) processors. They will only be available if support for the Intel 386 processor was specified at the time the compiler was built.

-fpu= yes **|** no
Specify whether operations involving floating point variables should generate in-line calls to a hardware floating point unit, or whether calls are made instead to library support routines. Using library support routines allows floating point operations to be carried out purely in software.

Default: **-fpu=yes**

N.B. We do not supply suitable library routines to do software emulation of floating point with the compiler.

-fpureturn =yes **|** no
This option is used to tell the compiler whether the library routines are such that floating point results are returned in the hardware FPU registers, or in normal registers. Note you should not normally change this value from the default unless you have been specifically advised to do so.

Default: **-fpureturn=no**

-peep= none **|** peepopt_list **|** all
Control the level of peephole optimisation that should be done. Past experience has shown that some of the more obscure bugs reported on the compiler are those where the peephole optimiser part of the compiler has made an invalid optimisation. You would therefore use this option if you suspect that the compiler has generated incorrect code, and you want to look at what would be generated if some or all of the the peephole optimisation was not done.

The meanings of the options are:

none All peephole optimisations are suppressed.

all All peephole optimisations are performed. It is equivalent to giving **-peep=flow** .

You can also exercise a finer level of control by specifying the exact combination of peephole optimisations that you want from the following options:

flow This tries to analyse the flow of the code to eliminate redundant loads of registers. A significant size savig normally results from this option. However if it goes wrong, the results can be rather unpredictable.

Default: **-peep=all**

If support for multiple assemblers and/or processors types was specified when the compiler was built, then a 386 processor target plus

a specific assembler can be specified using the following options:

-bas386
Generate 386 code. Use the syntax for Bruce Evan's 386 assembler for the output.

-gas386
Generate 386 code. Use the GNU 386 assembler syntax for the output.

-masm386
Generate 386 code. Use the Microsoft MASM assembler syntax for the output

-sysv386
Generate 386 code. Use the Unix SVR4 assembler syntax for the output.

INTEL 8086 OPTIONS

The options listed in this section apply when generating 16-bit code for use on Intel processors. They will only be available if support for the Intel 8086 processor type was specified at the time the compiler was built.

-fpu= yes **|** no
Specify whether operations involving floating point variables should generate in-line calls to the a hardware floating point unit, or whether calls are made instead to library support routines. Using library support routines allows floating point operations to be carried out purely in software.

Default: **-fpu=yes**

-peep= none **|** peepopt_list **|** all
Control the level of peephole optimisation that should be done. Past experience has shown that some of the more obscure bugs reported on the compiler are those where the peephole optimiser part of the compiler has made an invalid optimisation. You would therefore use this option if you suspect that the compiler has generated incorrect code, and you want to look at what would be generated if some or all of the the peephole optimisation was not done.

The meanings of the options are:

none All peephole optimisations are suppressed.

all All peephole optimisations are performed. It is equivalent to giving **-peep=flow** .

You can also exercise a finer level of control by specifying the exact combination of peephole optimisations that you want from the following options:

flow This tries to analyse the flow of the code to eliminate redundant loads of registers. A significant size saving normally results from this option. However if it goes wrong, the results can be rather unpredictable.

Default: **-peep=all**

-pointer= 16 **|** 32
Specifies that the code should be generated to conform to the small memory model (64K data + 64K code segments) which uses 16 bit pointers or the large model which uses 32 bit pointers.

Default: **-pointer=16**

If support for multiple processors and/or assemblers was configured when the compiler was built, then you can specify the target to be a 8086 processor and a specific assembler can be specified using the following options:

-bas86
Generate 8086 code. Use the syntax for Bruce Evan's 16 bit 8086 assembler for the output.

-gas86
Generate 8086 code. Use the GNU assembler syntax for the output.

-masm86
Generate 8086 code. Use the Microsoft MASM assembler syntax for the output

-sysv86
Generate 8086 code. Use the Unix SVR4 assembler syntax for the output.

TEXAS INSTRUMENTS TMSC30 PROCESSOR OPTIONS

The options listed in this section will only be available if support for the Texas Instruments TMSC30 DSP processor was specified at the time the compiler was built.

NOTE The TMSC30 support was developed by and is maintained by:

Ivo Oesch,
Selzweg 1,
3422 Kirchberg,
Switzerland.
email: b19oesch@isbe.ch (valid until March 1997)

-collect= yes **|** no
This option is used to control the level of effort that is put into removing redundant moves. The 'yes' value implies try harder.

N.B. This option is likely to be removed or combined with some other option in the future.

-delayed= n
This controls the condition under which delayed branches are used. The values of 'n' should be in the range 0 to 3. The meaning of the different vales is as follows:

0 No delayed branches are used

1 At least one useful instruction must follow to be able to use a delayed branch, or alternatively up to to 2 nops are allowed to be added to be able to use a delayed branch.

2 At least two useful instruction must follow to be able to use a delayed branch, or alternatively not more than 1 nop.

3 All three instructions following the delayed branch must be useful to be able to use a delayed branch.

The implementation of this option is done by taking 'useful' instructions from before the branch (i.e. the branch instruction is moved backwards in the generated instruction stream, and if this is not sufficient also moving instructions from the branches target and adjusting the target location accordingly.

-forcedsave= none **|** option_list **|** all
Forces the compiler to save specified registers on function entry and restore them when leaving a function. The 'option_list' may be any combination of the following register names:

none No registers are saved.

all All registers

You can specify specific registers by using any combination of the following (comma separated):

r0,r1,r2,r3,r4,r5,r6,r7
ar0,ar1,ar2,ar3,ar4,ar5,ar6,ar7
dp,ir0,ir1,bk,sp,st,ie,if,iof,rs,re,rc

In addition you can use **rn** to mean all r? registers and **arn** to mean all ar? registers.

It usually will only makes sense to use this option for special interrupt routines, so should not be switched on via the comand line.

The most likely way you would use this option is by including the following type of code sequence in your source:

```
#pragma forcedsave=all
special Interruptroutine
#pragma forcedsave=none
```

(This option was sdded since I needed it to get a realtime-operating system running, and it was needed for context-switching - I had to force a save of all registers onto the stack before switching the context (Reload SP and FRAMEPTR))

context (Reload SP and FRAMEPTR))

-mul32= yes **|** no
If enabled then real 32-bit multiplication is used for longs. If not enabled then the TMS320C30 24-bit multiplciation instructions are used for longs (and also any shorter integral type).

-optbranch= node **|** low **|** medium **|** hard
Control the effort that the peephole optimiser puts into optimising branch isntructions. The values have the following effects:

none No branch optimisation is done

low Only moving of blocks or replacing conditional jumps over loads with conditional loads.

medium In addition to the above, if the code before a branch instruction is the same as that before the target of the branch, then move the branch backwards and try to eliminate any resulting redundant code.

hard In addition to the above, try and common up any instruction sequences leading up to a branch to the same location.

-parallel= none **|** normal **|** all
*** DJW *** Not sure what this does but it is present in the code.

-peep= none **|** peepopt_list **|** all
Control the level of peephole optimisation that should be done. Past experience has shown that some of the more obscure bugs reported on the compiler are those where the peephole optimiser part of the compiler has made an invalid optimisation. You would therefore use this option if you suspect that the compiler has generated incorrect code, and you want to look at what would be generated if some or all of the the peephole optimisation was not done.

The meanings of the options are:

none All peephole optimisations are suppressed.

all All peephole optimisations are performed.

You can also exercise a finer level of control by specifying the exact combination of peephole optimisations that you want from the following options:

flow Used to enable the data flow analyzer. The data flow analyser will walk through the generated code keeping track of registers and attempting to replace each data access with a cheaper operation if possible.

CAUTION: This optimisation could lead you to violate a volatile constraint that you tried to apply at the C level. A work-around is to add a dummy 'asm' statement' something like asm("*Dummy, stops dataflow analyzer") before any statement which accesses operandas with volatile qualifiers. This stops the dataflow analyzer being able to do any replacements at this point.

Using this option can add significantly to the compile time - typically about a sixth.

pipeline This is used to control whether optimizations should be done that attempt to minimise pipeline conflicts arising from the usage of address registers as operands in instructions and in address generation. The optimisation involves re-ordering code sequences where possible to avoid such conflicts.

3operand Converts wherever it is possible two operand instructions into three operand instructions. This may open new paths for the other optimiser stages.

parallel Controls whether the peephole optimiser should attempt to use instructions that can be executed in parallel where possible. It involves replacing specific instructions with their parallel equivalents. Currently only ldi||ldi, ldi||sti and sti||sti combinations are supported.

remap Controls remapping of registers. For example an

remap controls remapping of registers. For example an

ldi rx,ry

is removed if ry can be replaced with rx in the following code or if rx can be replaced by ry in the preceeding code sequence.

This optimisation adds significantly to the compile time of programs.

debug *** DJW ***

Not sure what this does but is allowed for in the parameter options present in the code.

Default: **-peep=all**

-probe= no **|** yes
Specify whether stack probes should be generated when each stack frame is built. This option is only available if the **PROBES** option was configuration option was set when the compiler was built.

Default: **-probe=no**

-pseq= string
This option would rarely be changed as it is used for fine tuning of the peephole optimiser. It allows you to control both the number of passes made by the peephole optimiser, and also the specific optimisations that should be attempted in each branch. The 'string' passed as a parameter is in the form nnn/n...n/nnn/.../n where n is a number between 0 and 6. Each number controls specific optimisations that are done in that pass of the optimiser, and the slashes separate passes of the optimiser.

The meanings of the various numbers used in the string are as follows:

0 Do only simple standard optimisations

1 Combine forward. Arithmetic instructions with a following load is combined to give a 3-op instruction wherever possible.

2 Combine backward. Arithmetic instructions with a preceeding load is combined to give a 3-op instruction wherever possible.

3 Commutative. If a commutative arithmetic/locical instruction is followed be a load and the load can be avoided if the operands are swapped then this is done and the load deleted.

4 Do the optimisations that are controlled by the -remap option as longas it is active.

5 Do the optimisations that are controlled by the -collect option as long as it is active.

6 Do the optimisations that are controlled by the **-dataflow** option as long a it is active.

The default value currently built into the compiler is:

-pseq=54321/6/543210

If you find a sequence that give better results then please let Ivo know

-ramconst= yes **|** no
If enabled then large integer constants (more than 16 bits) are put into the **.const** segment which must be in the same data page as all other data segments. If not enabled, then large constants are constructed using ldi, shift and or instructions.

Whether this option is relevant or not will depend on whether you are constrained for space in the data page.

-shortfloat= n
Controls the cases in which short float constants are used according to the value of 'n' as follows:

0 Short float constants are only used if we are absolutely sure that

0 short float constants are only used if we are absolutely sure that they will not bring any loss in precision in the given constant.

1 Constants of the type 'float' are always represented in short form if they are in range. Constants of type 'double' and 'long double' only if there is also no loss in precision.

2 All floating point constants are represented in short form if they are in range (between -255 and +255) even if there is a loss in precision.

If support for multiple processors and/or assemblers was specified when the compiler was built, then you can specify the target to be a TI MSC030 processor and a specific assembler can be specified using the following options:

-rosc30
Generate code for the TI TMSC30 processor in the Rossin assembler format. This should also be compatible with the official TI assembler format.

ENVIRONMENT VARIABLES

If the compiler has been built to support environment variables, then the environment variable that corresponds to the name of that version of the compiler (i.e."C **386 ", " C86 ", " C68 "** or **" C30 "**) is checked to see if it is present, and if so is assumed to contain options in the same format as the command line options. This is done before processing the command line. Command line options will therefore over-ride the environment variable settings in the event of conflict.

The environment variable method is a very convenient way of setting defaults (such as the warning level) when you want a different one to the one built into the compiler.

EXIT CODES

The compiler returns the following error codes:

0	EXIT_SUCCESS. The compilation was successful. That is the source file was compiled, and there were no fatal errors.
other	EXIT_FAILURE. One or more fatal compilation errors were reported.

SUPPORT FOR #pragma DIRECTIVES

The ANSI C standard provides the **#pragma** statement as a way of allowing compilers to support non-standard (and typically non-portable) extensions to C. The support in the compiler for **#pragma** is behaves as follows:

a) If the text following the **#pragma** statement is valid for a command line option, then it is interpreted as being one. No check is made if this is sensible. A typical use for this facility is to perhaps temporarily turn up a warning level for a small section of the program. Another possible use is to dynamically change some of the code generation options such as the level of optimisation. If trying to take account of this facility please note that code is only generated when the end of a function is reached, and it is the settings for code generation at that point that are used. It is not possible to change such settings on a statement level basis.

b) If the text following the **#pragma** option is not recognised then the **#pragma** statement is simply ignored.

N.B. Please note that there is a high chance that we might change the above rules for **#pragma** support in future release of the compiler.

SUPPORT FOR asm KEYWORD

It is possible to build support for the 'asm' keyword into the compiler. This is, however, a very limited support in that it suffers from the following limitations:

- The text of the assembler code that is passed as a parameter to the 'asm' keyword is not syntax checked in any way - it is simply passed unchanged into the generated assembler file.
- If you want to reference a global variable then you need to add any prefixes (typically an underscore) to the names yourself.
- It is not possible to reference static or auto variables as these have internally generated labels.

We have no immediate plans to upgrade this support in any way. The use of the 'asm' keyword is completely non-portable and not part of the ANSI standard, so we do not feel the need to invest much work in getting it working. After all you can always write free-standing assembler routines that are added to your program at link time.

EXPLOITING COMPILER OPTIMISATIONS

This section discusses the optimisation methods used within the compiler and how you can code to exploit these too maximum advantage.

The philosophy that was used when developing the compiler was to try and strike a good balance between the efficiency of the optimisations that are done and the code/runtime penalties of doing the optimisations in the first place.

The decision was made to limit the optimisations that are will be done to those that can be done by pure static analysis of the generated code. More complex methods of optimisation have been avoided. The result has been a family of compilers that produce surprisingly good code without too much penalty in the runtime size or performance of the compilers.

To understand some of the following sections, you have to realise that the code generation of the compiler happens in two basic stages:

a) Generic code is generated that will work under all situations. No consideration is given at this stage as to whether the particular values of operands mean that shorter variants of instructions could be used. At this stage the following optimisations are performed:

- Allocating variables to registers
- Removing redundant stack updates.

b) The peephole optimiser is invoked that looks at the generated code to see how it can be improved. The optimisations that occur at this stage are:

- Choosing optimum code sequences.
- Commoning up repeated code sequences
- Eliminating redundant or unnecessary code.

The programmer can often increase the effectiveness of these optimisation processes by writing code appropriately.

Allocating Variables to Registers

The compiler will try and optimise the use of registers. You can stop this automatic allocation of variables to register by using the **-reg=no** runtime option to the compiler.

The compiler first allocates any variables for which the programmer has explicitly used the keyword **register** , and then (assuming there are still free registers) allocates further variables to registers using an algorithm that looks at how frequently they are referenced in the source program. This algorithm considers variables as suitable for holding in registers if they are referenced enough times so that the overhead of loading them into registers is less than the gains in code generation size of having them in registers.

This results in the following tips:

a) Avoid using the **register** keyword unnecessarily. The built in algorithms for allocating variables to registers are very good, and often will achieve better results than the programmer.

b) Consider assigning variables used in loops explicitly with the **register** keyword. Because only static analysis techniques are used, the compiler optimises for space, and may not realise the run time performance advantage of keeping loop variables in registers (albeit possibly at the cost of increasing code size).

Removing Redundant Stack Updates - "Lazy stack updates"

If there are several calls to functions without any intervening transfers of control, then the compiler can accumulate the stack tidying operations normally performed after such calls and do them all at as late a stage as possible. This means that multiple small stack adjustments can be replaced by a single larger one (or even sometimes not do it at all if the end of a function is reached first). This optimisation results therefore in both size and speed gains.

There are times, however, when it is inadvisable to do this optimisation. You can therefore exert tight control over exactly this optimisation by using the **-stackopt=xxxx** runtime option. The values of xxx have the following effect:

safest
This disables this optimisation completely. This is advisable if you have routines which make any significant number of recursive calls (either directly or indirectly via other routines). This is because it is likely that there will be obsolete parameters left occupying space at the point of recursion. This can cause excessive stack usage if the recursion is to any depth.

minimum

minimum

This is the safest form of stack optimisation and is the default compiler operation. With this option, stack optimisation is done unless a function call is found which is to alloca(), a function whose name starts with an underscore, or a function that is being called indirectly via a function variable (which means its name is indeterminate). This behaviour is to allow for the occasional routine (typically an assembler routine in a library) that directly manipulates the stack and can return with the stack set to a different value to that on entry. Note that standard C routines cannot exhibit this behaviour.

average

This option allows for optimisation of calls to routines that contain an underscore. Its behaviour is otherwise as described for **n=minimum** . This option can have significant gains in the situation in which underscores are being added to the user defined names for the purposes of name hiding within libraries.

maximum

This option allows for optimisation of calls to routines that are called via a function variable (and whose name is therefore indeterminate). This level of optimisation can have a larger gain than is at first apparent. This is because the C68 optimisation for the use of registers can result in the address of a frequently called function being held in a register variable. This level of optimisation allows the lazy stack optimisation to be applied to such calls as well. This level of optimisation should be safe for pure C code. However, it is not the default as it is very difficult to track down problems arising from doing lazy stack optimisation when it is incorrect to allow it.

Choosing Optimum Code Sequences

This optimisation is simply a case of examining the code generated looking for common code sequences that can be replace by faster and/or shorter ones. This level of optimisation can be disabled by using the **-opt=no** keyword. However, there is normally little to gain by disabling this optimisation unless you suspect an error as it has little detrimental effect on compilation speed.

Commoning up repeated code sequences

The compiler will attempt to common up repeated sequences of code within a function. This can result in significant reduction in code size. However, as this optimisation can impose a significant time penalty on the compilation process, it is only invoked if the **-O** runtime option is supplied to the compiler.

To maximise the potential gains that will be achieved by this optimisation the following tips may be useful:

a) Try and ensure that the code sequences leading up to return statements or break statements within a switch construct are the same. This will allow the compiler to only generate the code once and implement all repeated occurrences of such code as simple branches to the first one.

b) If you have such sequences that simply differ by one variable, then it may be worth assigning that variable to a temporary one and using that if as a result a larger sequence of code is common.

Removing Redundant or Unreferenced Code

This optimisation is done only if the **-O** runtime option to the compiler was used. It looks for any code sequence that cannot be reached. If the code in question was a direct result of the way the programmer wrote the source code then, if level 4 warnings are active, appropriate warning messages will have been output during the parsing stage. However, this situation can also arise as result of the effects of previous stages in the optimisation process.

KNOWN BUGS AND LIMITATIONS

The following are known bugs in the 4.5 release of the compiler.

- Adjacent wide string literals are not concatenated.

The following undefined behaviours are not detected:

- An attempt is made to modify a string literal of either form.
- An object is modified more than once, or is modified and accessed other than to determine the new value, between two sequence points.
- The value of an uninitialised object that has automatic storage duration is used before a value is assigned.

ANSI FEATURES NOT SUPPORTED

The following features specified in the ANSI standard are NOT supported

The following features specified in the ANSI Standard are NOT supported

- Trigraphs. It is possible, however, that you have a pre-processor that handles trigraphs, in which case this is done there rather than in the compiler program.

ANSI EXTENSIONS SUPPORTED

If the **-extensions=yes** run-time option is used and the **EXTENSIONS** configuration option was set when the compiler was built, then the following additional functionality is supported. These are based on the proposed amendment to ANSI C that has not yet been ratified.

- The C++ style of comment is allowed (i.e. those starting with the // sequence).
- The '**restrict**' keyword is recognised and the associated syntax rules for restricted pointers.
- Other new reserved words such as '**class** ', '**private** ' and '**public** ' are recognised and flagged as errors when used as variable names.

CHANGES TO FEATURES IN K&R COMPATIBILITY MODE

If K&R compatibility mode is specified by using the **-trad=yes** run-time option, then the following changes occur in the features supported by the compiler:

- The **long double** qualifier is not allowed.
- The use of the **long float** qualifier as a synonym for **double** is permitted.
- The **const** keyword is not allowed.
- The **volatile** keyword is not allowed.
- The **signed** keyword is not allowed.
- String concatenation is not performed.
- Single copies of identical strings are not generated. Instead separate copies will be generated every time a string is used.
- ANSI style function prototypes are not allowed.
- ANSI style function declarations are not allowed.

ERROR AND WARNING LEVELS

The errors and warnings within the compiler are classified into various severity levels. The higher the level, the more pedantic the level of messages that are output. By default all messages with severity 0 are errors, and all those with higher levels are merely warnings. The **-warn=n** and **-error=n** runtime parameter options allow the user to vary the default treatment of these levels.

The compiler is normally supplied with warning level 3 set as the default warning level (if not changed via the command line or in an environment variable). It is good practice to try and write code that compiles without warnings even at levels 4 or 5. There are then less likely to be subtle bugs lurking in your code that are coding mistakes that are difficult to spot. A real zealous coder will definitely want to achieve level 6, and possibly level 7. You have to be a zealot to want to expend the effort required to get code to compile warning free at level 8.

The levels currently supported are as follows:

0 Messages at this level are always errors. If you specify this as a warning level, then effectively all warning messages are disabled.

1 These are severe warnings that should not normally be suppressed. They typically relate to problems at the code generation stage of the compiler or to constructs which only some compilers will allow.

2 These relate to problems with the code that normally indicate problems or potential problems. They are typically easy to fix - normally by adding a cast or something similar.

3 This level relates to warnings that are commonly encountered when porting code. The warnings at this level may not indicate an error, but they should certainly be checked out.

4 This level of warning indicates problems that are often encountered in porting, but that are probably not an error. It is still a good idea to get your own code to compile cleanly at this level of warning as it will minimise problems later.

5 This level of warning is for short cuts that experienced C programmers often use, but that are occasionally done in error. You are most likely to find this level useful when trying to track down an error that you are having trouble locating. It is good practice to write code that is warning free even at this level.

6 This level is very strict. It is primarily intended to help spot code that might cause problems if you intend to port the program to another machine or compiler.

7 This is an extremely pedantic level. It is intended to allow you to help you write extremely "clean" code. It will also help with porting programs although the warnings generated at this level are for items that have been found to be less likely to cause problems than those reported at level 6.

8 This mode is extremely strict. So much so, that it is not always possible to write the code in such a way as to completely eliminate all level 8 warnings.

ERROR AND WARNING MESSAGES

The following is a list of the error messages that can be output by the compiler. In most cases the messages are self-explanatory, but where this is not so, additional information is given about the possible cause of the error message.

Where variable information can be inserted into the message, then this has been specified using the **printf** format string method.

LEVEL 0

This level of message is always an error. It is not possible to make the compiler treat such messages merely as warnings.

& operator may not be applied to bitfields
The ANSI standard does not allow the address operator to be aplied to bit-fields.

& operator on register variable '%s'
The ANSI standard does not allow the & operator to be used on variables that have been qualified with the **register** keyword.

{ expected on initialiser
If you are initialising a complex structure such as an array or structure, then the initialisation values should be enclosed in braces.

an object type expected
A reference to an object was expected but not encountered. This could, for example, be generated by attempting to increment/decrement a pointer to a function.

arithmetic type expected
an integral type (long, int, short or char) or a floating point type(float, double or long double) was expected.

break not allowed here
A break statement was encountered when not in a do, while, for or switch statement.

cannot nest function definition '%s()'
The ANSI C standard does not allow function definitions to be nested.

cannot subtract a pointer from an integral value
It is only allowable to subtract an integral value (long, int, short or char) from a pointer, and not the other way around.

case not allowed here
A case statement has been encountered when not within a switch statement.

character constant unterminated or too long
Either the terminating quote character was missing from the character constant or else there were too many characters within the character constant.

constant expression expected
During a variable initialisation an expression was encountered which is not a constant expression.

constant expression exceeds representable range of type '%s'
This will normally occur when you try and either assign or initialise a variable with a constant that is outside the range that will fit in the given type.

constant integer expression expected
During a variable initialisation an expression was encountered which is not a constant integral expression.

continue not allowed here
A continue statement has been encountered when not within a do, while or for statement.

declared argument '%s' missing
A K&R function definition has an entry in the parameter definition list which is not in the parameter list of the function.

duplicate case label 'case %ld'
A case statement has been encountered for a value which has already been associated with a previous case statement inside the same switch statement.

statement.

duplicate default label in case

A default label has already been encountered inside the switch statement. Only one such label is allowed.

duplicate label '%s'

The label has already been found within the current block.

enumeration constant too large

An enumeration value has been defined which is too large to fit within an 'int' type.

error dereferencing a pointer

An attempt has been made to derefence an object that cannot be dereferenced. An example might be to try ***i = 3;** where i is an integer.

error doing a cast

An attempt to perform an illegal cast operation. An example might be an attempt to cast a structure to a structure of a different size.

error while scanning a parameter list

This implies that the compiler has encountered something unexpected while scanning a parameter list. It is commonly caused by a misplaced comma, or a mispelled type keyword.

expression expected

An expression was expected and not encountered. This can happen, for example, if the condition in an 'if' statement is missing.

extern definition of '%s' redeclared static

You have earlier declared as globally visible a function or variable that you have now defined as static and therefore limited to the current scope.

floating point constant expected

An attempt was made to initialise a floating point variable with an expression that could not be evaluated to a floating point constant.

function declarator not allowed here

This can be encountered if an attempt is made to write a function definition which returns a function - it is only possible to return a pointer to a function.

function returning array type

A function is not allowed to return an array type. It can only return a pointer.

function type expected

An attempt has been made to call a function by using an variable which is not a function pointer.

function '%s' declared but never defined

This will occur if you put a forward declaration for a function in a file, and then never define that function. It could also occur if you meant to forward declare a library function, but omitted the 'extern' storage class specifier.

function '%s()' default promotion / prototype mismatch

This is typically caused by mixing ANSI and K&R methods of function declaration and definition. This is of particular importance for functions which have parameters of types 'char', 'short' or 'float' as the parameter promotion rules for these types are different for K&R and ANSI declarations and definitions.

function '%s()' mismatched number of arguments

The number of parameters does not agree between two different declarations for the same function.

function '%s()' prototype mismatch

This indicates that the for the specified function, there are incompatible definitions or declarations. This can be either in the type returned, or the number or types of the parameters.

general error

This error means that a consistency check within the compiler has failed. Please report the circumstances that caused the problem, and ideally provide a sample of code that can be used to reproduce the problem. It is preferable if any code that is supplied to illustrate a problem has already been passed through the C pre-processor. This eliminates any dependencies on system specific header files.

identifier expected

The name of an identifier was expected but some other token was found instead.

identifier list not allowed on function declaration

A function declaration has been encountered which has a K&R style paramter list. Such a list is only valid on function definitions and not function declarations.

illegal cast from '%s' to '%s'

You have specified a cast operation between to types that are not cast

compatible.

illegal character '%c'
A printable character has been encountered in the source which is not legally allowed in any C token.

illegal field width
You have specified a width to a bit field that is too large. ANSI restricts bit field widths to being no larger than that of the 'int' data type.

illegal initialization
The compiler has recognised that you are trying to initialize a variable, but the type of initialization you are trying to do is not permitted.

illegal redeclaration of '%s'
The function/variable has been declared in a way that is incompatible with an earlier use.

illegal 'sizeof' operation
An attempt has been made to take the size of an item that does not have a size attribute. An example might be to try and take the sizeof a function name.

illegal storage class
A storage specifier has been used multiple times or else in an inappropriate place.

illegal type combination
Type specifiers have been used in a combination which is not valid. An example might be to try and use "short char".

illegal unprintable character (value=0x%x)
An unprintable character has been encountered in the source which is not legally allowed in a C source value. As it is unprintable the hexadecimal value that corresponds to its internal representation is given in the error message.

"implicit conversion to pointer on register variable '%s'"
Self explanatory.

"incomplete '%s' declaration"
Self explanatory.

"initialization invalid"
Self explanatory.

"integral type expected"
A variable with an integral type (long, int, short or char) was expected.

"l-value required"
A l-value is simply an expression which it is legal to have on the left side of an assignment expression. This means that you have an assignment (or an implicit assignment) where this is not true.

"modified 'const' value"
An attempt has been made to change the value of an object that was declared as 'const'.

"parameter count incorrect for function %s"
The number of parameters passed in the function call does not agree with the number that is specified as required in the function prototype.

"pointer type expected"
Self explanatory.

"problem with pre-processor output"
This indicates that what looks like a preprocessor symbol (one starting with #) was found in the source file, and it was not one that the compiler expects to get past the pre-processor. This is typically caused by trying to use the compiler on raw C source before it has gone through the C pre-processor.

"qualifier already specified"
This means that there are duplicate qualifiers of the same type referring to the same variable or function declaration/definition. The second one will simply be ignored, but the source should be corrected.

"qualifier mismatch"
When comparing two type definitions the 'const' or 'volatile' qualifiers do not match.

"'restrict' only allowed on pointer types"
The 'restrict' qualifier can only be applied to variables that are of pointer type.

"return expression of type void"
It is not possible to return an expression which evaluates to type **void**
.

"return value specified to void function"

A return statement has been found that is attempting to return a value for a function that was defined as returning void (i.e. no value returned).

"scalar type expected"

A type which is an integral type (long, int, short, char) or a floating point type (float, double, long double) or a pointer was expected.

"string constant unterminated or too long"

This message may well occur well after the point at which the string constant started. It is quite often caused by mismatched comments or **#if** / **#endif** directives.

"tag usage '%s' mismatch"

An attempt to use a struct, union or enum tage more than once but applied to a different type than that used in the original use.

"too many initializers"

The number of initializer values would exceed the size of the variable space allocated to hold them.

"type specifier '%s' already specified"

A type specifier has been used more than once. An example might be: **int int i;**

"type mismatch error"

When comparing two type definitions for compatibility they did not match.

"type/operand has unknown size"

An attempt has been made to use the size of a type when the type is an incomplete type and therefore has not size information available.

"undefined identifier '%s'"

An attempt to use an identifier before it has been defined. A common cause is that the name has been misspelt.

"undefined label '%s'"

A goto statement is attempting to go to a label which has not been defined within the current scope.

"unexpected end of file"

This is typically caused by a mismatch between the number of start and close braces.

"unexpected symbol '%s' found"

This simply means that the symbol shown was not legal at this point, and the compiler has been unable to specify the error more accurately.

"value of escape sequence out of valid range"

The backslash escape character has been used to define a character constant with a value that is too large to fit into the range of values that are legal for a character.

"visibility specifier '%s' only allowed with 'class'"

You can only use this type of visibility specifier in conjunction with a class declaration or definition (ANSI extension).

"void parameter is passed to function %s"

An attempt to pass a parameter which has a type of 'void'. This is not allowed.

"'%s' is not a struct/union member"

You have used the specified variable name in a context in which a structure or union member name is required, and the name is not defined as being part of the structure or union in question.

LEVEL 1

This level of message is used to indicate code that although allowed by C is extremely bad coding practice, and as a result is normally not what the programmer meant.

"bit field type should be unsigned or int"

The type for a bitfield should be of type int or unsigned int. Some compiler allow other types (such as short) but this is an extension to ANSI and is not portable.

"extern definition of '%s' redeclared static"

Self explanatory.

LEVEL 2

This level of warning is used to indicate code that may well not be an error. However, experience has shown that in reality the code does not perform the action that was intended.

"conversion between incompatible types '%s' and '%s'"

This message indicates that the two types in question are not defined by the C standard to be compatible. If you really mean the statement, then the message can be suppressed by use of a suitable cast.

"format string for %s() incorrect"
This indicates that the format string for a format string for a routine from the specified printf/scanf family of routines is incorrect. Typically this means that there is a **%** symbol that is not followed by a legal conversion character.

"size of parameter %d changed by prototype on function %s"
This implies that an implicit cast was applied as a result of a prototype being in scope. Care would need to be taken when porting such code to an environment which does not have an ANSI compatible C compiler. It is often a good idea to add an explicit cast to such calls as this at least makes it clear what is happening, and will make code more portable.

"'sizeof' value is zero"
"'sizeof' value %d is greater than '65535'"
This will occur when the size of a **sizeof** operator is set to be only 16 bits, and the result of a **sizeof** operator is larger than 16 bits. The data type returned by the **sizeof** operator is in fact determined by the value defined for **TP_SIZE** in the configuration file (config.h) used when **c386/c68** was compiled. It is important that this value should agree with the value defined for **size_t** in your system include files.

"\x not followed by any hex characters"
The \x sequence that ANSI specifies as being used as an escape sequence to introduce a hex character was not followed by values that could be interpreted as hex.

LEVEL 3

This level of message indicates code that is probably not an error, but is untidy. Messages in this category can normally be suppressed by making simple modifications to the source code.

"constant %ld not within range of type '%s'"
You have tried to assign a constant to a variable that is too large to fit into a variable of that type. An explicit cast will eliminate this warning, but a better solution is to change either the data type or the constant so that the warning is no longer relevant. Note that there is one case where you sometimes get an unexpected complaint about a negative constant being out of range. This occurs when you use a bitwise not operator on a signed field. This is potentially non-portable. The recommended solution is to only use this operator on unsigned fields or unsigned constants (so you can normally just add a U to the end of the constant to make it usnigned).

"conversion between incompatible pointer types"
Very common message when a pointer of one type is assigned to a pointer of a different type. Inserting the relevant cast will suppress this message.

"dangerous truncation of pointer to '%s'"
You have tried to store a pointer in an integral type that is not large enough to hold pointers without the risk of losing information. This is typically because a programmer has made the assumption that the size of a pointer is the same as sizeof(int). If you mean it then add an explicit cast to stop this warning being generated.

"division by zero"
You have tried to divide an expression by a zero constant. This is typically because a more complex expression, possibly involving pre-processor macros, has evaluated to zero.

"dubious %s declaration; use tag only"
This normally means that a structure or union pointer has been encountered using a tag which has not been defined. This can often happen when a tag is encountered for the first time in a function prototype. As this tag goes out of scope at the end of the function prototype this means that you can never call the function with a parameter of the correct type. To avoid this problem either the structure definition must precede the prototype, or you must forward declare the structure type before the prototype.

"escape ignored in sequence '\%c'"
The character following the \ is not one that is supported as a valid escape sequence. The effect is that the \ character is lost, and the next character is handled unchanged.

"function '%s' declared but never defined"
This normally means that there is a forward declaration for a static function, but that the code defining that function is not present.

"implicitly declared function: 'int %s()'"
This means that there is no declaration (either ANSI or K&R) in scope for this function. If the function is a standard library function, then it means that the relevant header file has not been included.

"no value specified in 'return' statement"

"no value specified in 'return' statement"
This occurs when a return statement is found for a function that has an implicit **int** type. It can be suppressed by defining the function to be of type **void** .

"parameter before ',...' causes undefined behaviour"
The last parameter before a varardic parameter list is of a type that may cause undefined behaviour. This is because the type of that parameter is such that it cannot safely be used within the macros defined in the stdarg.h header file.

"qualifier inconsistent with type 'void'"
This implies a const or volatile qualifier used in conjunction with a void type.

"redeclaration of '%s'"
The define variable or function has been defined more than once. This is typically because it is defined in multiple different header files. It is a good idea to try and set up header files so that each variable or function is only defined in one place to avoid any potential confusion that might later arise if you change one declaration and not the other one.

"returning address of a local variable"
You have returned the address of a local variable (i.e. one on the stack). This is very unlikely to be what you meant to do.

"using out of scope declaration for %s"
This means that an externally linked routine or variable is used outside the block in which it was declared.

This is commonly caused by using routines for which the correct header file has not been defined as this causes an implicit declaration at the first usage, and then this message subsequent functions which use that same routine.

"'%s' is always positive"
This message occurs if you try and test and unsigned value for being a negative value (i.e. less than zero). This does not make sense, so is almost certainly a logic flaw in your program.

LEVEL 4

Messages at this level are not strictly speaking errors, but they do indicate code that could be improved. In particular, they indicate code that might have portability problems.

"& operator on function ignored"
The & operator was specified on a function reference. It is not required as it is implicit.

"%d expression to '?:' operator cast to void"
You appear to be throwing away the specified result. Did you meant to?

"argument '%s' implicitly declared 'int'"
This means that an argument to a function has been specified which has not been explicitly given a type. It has therefore been treated as an int. Declaring the argument type explicitly will stop this message being generated.

"array type used in '%s' statement"
An array type was used as the condition for an 'if' or 'switch' statement. Although legal this will almost always not be what was intended.

"definition of '%s' hides an earlier definition"
This occurs when a variable name is used in an inner block that has the same name as one that has a wider scope. It is just a warning that during the duration of the block the variable at the outer level will be inaccessible.

The commonest cause is when the name of a parameter to a function is the same as that used for a global variable.

"empty statement"
An empty statement has been found following a construct like an **if** or **while** statement.

There are situations in which this is exactly what the programmer meant, but it might also be due to an accidental semicolon being present.

If you meant to have an empty statement then a way to eliminate this warning is to simply put a statement at the appropriate place of the form:

(void)variable;

(void)variable;

The cast to void will mean that the optimiser will ensure that no code is generated, but the presence of the statement tells the compiler that you know there is not a missing statement or extra semi-colon.

"function '%s' redeclared, assumed static"
Self explanatory.

"if statement has no effect"
The if statement has an empty statement in the result branch. This does not normally make much sense, so you probably did not mean it.

"implicit cast of pointer loses const/volatile qualifier"
An assignemnt of a variable which has a 'const' or 'volatile' has been made to a variable which doe not have the corresponding 'const' or 'volatile' qualifier.

"K&R style function"
This message will only be output if the **-obsolete=yes** runtime option to the compiler has been used. It is a warning that in the future that support for K&R style function definitions may be removed from the ANSI C standard.

"parameter %d to function %s() promoted to '%s'"
This means that the size of a parameter was changed according to K&R promotion rules. This message can be suppressed by having an ANSI style prototype of function definition in scope, or by using an explicit cast.

"pointer difference between different pointer types"
You have subtracted to pointers of different types. This construct is potentially non-portable. The portable way is to cast both pointers to long before doing the subtraction.

"shift by %d" outside range of '%s'"
You have attempted to shift a value by more than the number of bits in the field which will always result in zero. Did you mean this?

"statement not reached"
This message means that the statement in question is preceded by a construct that means program flow cannot reach the statement.

A typical cause might be code that follows a return statement without a label. This can quite often happen in the more subtle context of a switch statement in which all cases are terminated by return statements, but there is then code following the end of the switch statement.

"storage specifier not at start of definition"
The ANSI C standard has declared that a future version of the standard may require storage specifiers to be used only at the start of definitions. The current version of the ANSI C standard allows more leeway.

LEVEL 5

"! operator used with a constant expression"
It is very unusual to use the not operator with a constant expression - you can always rewrite such an expression to eliminate the need for the not operator. It is much more likely that you really meant to use some other operator.

"'%s' has 'const' qualifier but is not initialised"
As you can never change a variable of const type it does not make much sense not to initialise it. Another common mistake is that you meant this to be a declaration of an external variable but you omitted the extern keyword.

"'%s' modified and accessed between sequence points"
The standard for the C language allows the compiler implementor some latitude about the order in which expressions are evaluated, but also defines very carefully the sequence points at which the programmer can assume the result has been calculated.

If you use a construct that both modifies a variable abd accesses its value between such points, then the result is implementation defined and therefore almost certainly non-portable.

"'%s' modified more than once between sequence points"
The standard for the C language allows the compiler implementor some latitude about the order in which expressions are evaluated, but also defines very carefully the sequence points at which the programmer can assume the result has been calculated.

If you use a construct that modifies a variable twice between such points, then the result is implementation defined and therefore almost

certainly non-portable.

"assignment of negative value to '%s'"

You have assigned a negative value to an unsigned type. This means that the value will simply be stored using the bit pattern of the negative number and will normally result in a large value being stored. If you meant this and want to suppress the warning simply add an explicit cast.

"assignment in conditional context"

This means that there was no conditional test found, so it is possible you put an assignment when you meant to put an equality test. This message can be suppressed by testing the result of an assignment against zero.

"dangling 'else' statement"

This is a warning that a construct of the form

```
if (test)
...
else
if (test2)
...
else
```

has been encountered, and it is possible that the last 'else' statement is not associated with the if statement that the programmer mean. Use of braces to clarify the statement will suppress this warning.

"format mismatch with parameter %d on function %s"

This message is output when checking format strings for the 'printf' and 'scanf' families of routines against the following parameters. This indicates the parameter is not of the type indicated by the format string.

"ignored return value from function %s"

This means that you did not use the return value from a function. Inserting a (void) cast before the function call will suppress this message.

"label '%s' declared but not used"

A common cause of this can be leaving the 'case' keyword of a branch of a switch statement. This can be remarkably hard to spot sometimes as the code is still syntactically correct.

"mismatch on storage specifier"

The function definition has a different storage qualifier on a parameter than the prototype for the function. Typically this is the inconsistent use of the **register** keyword. This is currently allowed under the ANSI C standard, but not recommended.

"no prototype defined on called function %s"

This occurs if the function has been earlier defined via a K&R definition, and there is no ANSI prototype in scope.

"no value specified in implicit 'return' statement"

The end of a function definition has been reached so that there is an implicit return. The type of the function is not void so in theory there should be an explicit return statement with a value. However, much C code is written so that the type of a function is defaulted (which means it becomes int) and the return value of a function is not used. Explicitly declaring the function type as void will stop this message being output.

"result of expression has been discarded"

You have asked the compiler to calculate something and then never used the result. This code will therefore be ignored.

"unary '-' applied to unsigned expression"

The expression is unsigned, so if the result would be negative you may not get the result you expect (it will become a large positive number!).

"variable '%s' may be used before set"

It appears that you have used the above variable before you have assigned a value to it.

Sometimes this will happen in loops and it may not be obvious how to suppress the message.

"variable/function '%s' not used"

There is a variable and/or function that has been declared but not used.

This check is done at the end of a function/block. This means that for a variable, the line number quoted with this message is that for the

brace at the end of the block that defines the unused variable. For an unused static function, the line number quoted will typically correspond to the end of the source file.

LEVEL 6

The warnings that occur at this level are not normally relevant to the average user.

"a cast from '%s' to '%s' loses accuracy"
This is really not a problem if the action is what was intended. You can eliminate the warning by putting in an explicit cast.

The purpose of this warning is to highlight situations in which there may be an implicit assumption built into the code as to the size of a field of a particular type, which may not be true on the current machine.

"constant promoted to '%s'"
A constant has been implicitly promoted due to the way it has been used. You can avoid this warning by either making sure the constant is of the right type or adding a cast.

"expression involving floating point"
There is an expression that involves floating point, and you are working with a version of the compiler that recognises the keywords for floating point, but that is not able to generate code for floating point.

"implicit cast of '%s' to enumeration value"
An integral type (long, int, short, char) has been assigned to a variable which is of an enumeration type. You can add an explicit cast to eliminate the warning.

"initialisation incomplete - remaining fields zeroed"
This message is output if the initialisation statement supplied for a data item would not initialise all elements of that item.

There are often times when this is exactly what the programmer meant to do, but occasionally it is due to the initialisation being incomplete.

"parameter before ',...' causes undefined behaviour"
Technically this is the same warning as the message with the same text that is output at warning level 2. We move the warning to level 6 when the parameter in question is a function pointer because this is actually more likely to give the expected behaviour than the other types that cause the level 2 version of the message.

"possibly unnecessary cast from '%s' to '%s'"
You have added some explicit casts that seem to be unnecessary and may result in redundant code being generated.

"use of 'char' is possibly non-portable"
The ANSI standard allows the 'char' data type to be either signed or unsigned as an implementation defined decision. You should therefore be wary of making assumptions about whether characters or signed or unsigned if you want to write code that is portable between different machines, or even different compilers on the same machine.

"use of 'char' as array index is possibly non-portable"
The C standard leaves it up to the implementor whether the 'char' data type is signed or unsigned. You can eliminate this warning by either using a different data type or adding an explicit cast to either 'signed char' or 'unsigned char'.

LEVEL 7

The warnings that occur at this level are not normally relevant to the average user. They are extremely pedantic in nature and are normally only really relevant to tidying up the code.

"C++ keyword used"
This says that you have used a name for an identifier that would be a reserved word with a C++ compiler.

"constant expression used in '%s' statement"
A constant expression has been used for the condition in an 'if' or 'switch' statement. This does not really make much sense. This warning can help pinpoint the situation in which the condition test is not quite what you meant it to be.

"function not using ANSI style parameters"
A function has been found that is using K&R style methods of declaring its parameters. ANSI have declared their intent to remove support for this construct in future releases of the ANSI C standard.

"implicit cast of 0 to pointer type"
This occurs when the constant zero has been used in a circumstance

(normally as a parameter to a function) in which a pointer type is expected. The ANSI standard specifically allows zero to be used in such circumstances without an explicit cast to a pointer type as an equivalent to the NULL pointer type. However, most modern systems will define NULL using something like:

#define NULL ((void *)0)

in which case NULL can be used instead of zero when you really mean it which will stop this warning from being output.

"partially elided braces on initializer"
This rather cryptic message can be output when initialising unions, arrays and structures. The C standard says that initialisers for all such constructs should ideally have braces around them. This message therefore means that the bounds of a particular element of the data structure had to be deduced from its position in the initialisation list rather than being explicitly bounded by braces.

The requirement to suppress this message is that the values for an union, array or structure must start and end with braces. In the case of more complex structures such as an array of structures there must be braces around the whole set of values (ie the array) and also braces around the values for each occurrence of the structure.

"signed types with bitwise operator possibly non-portable"
ANSI states that if you try to do bitwise operations with negative number, then the result is implementation defined. The implementation is free to decide on whether the sign bit is propagated or not. Such code will therefore somitemis give different results on different compilers.

"switch has no 'default' statement"
It is always a good idea to have a default statement in all switch constructs. If you do not expect to get there, then simply include a line of the form

```
assert(0);
```

as the operation to be performed. That way you will pick up any logic errors which result in the default branch unexpectedly being taken.

"unnecessary cast to 'void'"
This is when a void expression is explicitly cast to a void. This is a null operation, so you do not need to specify the void.

LEVEL 8

The warnings that occur at this level are not normally relevant to the average user. They are extremely pedantic in nature and are normally only relevant to those who are writing code that has to conform to the very highest standards - perhaps for applications that are safety critical as an example.

It can be very difficult to eliminate all warnings at this level. As a result, whether the warnings at this level are even output at all is determined by the configuration options set at the time the compiler is built.

"%s has already been declared"
You have declared the function or variable more than once. The definitions are the same so this is harmless, but you might want to see if you can remove one of the declarations to avoid any potential future problems where you change one declaration and not the other one.

"'%s' has not been previously declared"
This will occur if the first time the compiler comes across an externally visible function is when it is defined. It is good practise to have declarations of all such functions used in a shared header file if they are not.

"implicit cast from '%s' to '%s'"
This occurs when an assignment or expression evaluation generates an implicit cast. There are times when due to the way the compiler works it will not be possible to eliminate this warning.

AUTHOR(s)

Versions prior to release 4.0:
Christoph van Wullen.

ANSIfication work and other enhancements in Release 4.0 and later releases:

Keith Walker

email: kdw@oasis.icl.co.uk
(bug fixes, IEEE support, ANSIfication)

Dave Walker
email: d.j.walker@x400.icl.co.uk
(IEEE support, Errors/Warnings, documentation)

TMSC30 support:

Ivo Oesch
Selzweg 1, 3422 Kirchberg, Switzerland
email: b19oesch@isbe.ch (valid until march 1997)

CHANGE HISTORY

The following is the change history of this document (not the compiler itself). It is intended to help users who are upgrading to identify the changes that have occurred.

12 Jun 93 Added full list of error messages that can be output by the compiler.

10 Jul 93 Added specification of new **-frame** parameter option for C68 variant.

10 Oct 93 Checked that list of error/warning messages corresponds to those actually in C68 v4.3, and expanded some of the explanations.

19 Mar 94 Updated to add new parameter types for C68 Release 4.4 and also updated the lists of error and warning messages.

Major Revision

28 Apr 94
- Major changes to the section that talks about optimisation methods.
- Updated lists of error and warning messages bring it in line with those that can now be output by the compiler.

21 May 94
- Added descriptions of -align and -packenum.
- Updated lists of warning messages.

10 Nov 95
- Updated list of error and warning messages.
- Merged in known bug list.

24 Nov 95 Added description for **-prefix** runtime parameter option.

Major Revision

07 Sep 96
- Updated all parameter descriptions to conform to the new syntax.
- Added descriptions of new parameters that have been added.
- Re-ordered the options description to more clearly show which options are only relevant to particular target processor types.
- Updated lists of error messages and warning messages to bring it in line with current compiler version.

04 Oct 96 The -short and -small runtime options renamed to -int and -pointer respectively, and the list of valid options changed.

16 Nov 96 Documented changed options to the **-peep** parameter, and various larger scale changes within the TMSC30 specific parameters.

10 Dec 96 Documented new **-interrupt** option for use with the 68K code generator.

AS68 Assembler

NAME

AS68 - assembler for use with C68 system

SYNOPSIS

AS68 [options] input_file output_file

DESCRIPTION

The assembler used by the C68 system is a derivative of the "Sozobon" public domain JAS assembler. It has been ported to QDOS and modified to produce SROFF output.

The JAS assembler was designed for compatibility with the Alcyon assembler. It, and thus AS68, do not provide many features the assembly language programmer might want. AS68 is intended more for use by a compiler front-end. AS68 generally produces smaller code than the Alcyon assembler because it is smarter about generating short branch instructions. Also, AS68 uses no temporary files and runs quite a bit faster than Alcyon.

Some of the command line options are accepted for compatibility with the Alcyon assembler, but are actually ignored. The following command line options are supported:

= <number>
Set the stack space. This should not be needed with C68 Release 2.00 or later.

% <number>
Set the heap space. This should not be needed with C68 Release 2.00 or later.

-N
Do not generate short branch instructions.

-V
Print a version message.

-l
Ignored.

-u
Ignored.

-s dir
Ignored.

-L n
By default, no local symbols are placed in the symbol table of the output. This option instructs AS68 to put all symbols into the symbol table if **n** is 2 or greater. If the option **-L1** is given, symbols whose name does not start with 'L' (ie internal labels generated by the compiler) are written to the symbol table.

DIRECTIVES

The following is a list of the directives supported by the assembler. This version of the assembler is compatible with source intended for the Alycon Assembler, and also for the MINIX ACK assembler. Many of the directives therefore exist in two forms.

.align n
Position on an boundary that is a multiple of **n** . Typically used to round to multiples of 2 or 4.

.ascii
Define a series of ASCII string (not zero terminated).

.asciz
Define a zero terminated ASCII string.

.bss
Start the BSS section.

.comm
Start of a common section

.data
Start the DATA section. Equivalent to ".sect data"

.dc.n
Define data elements. The size of the elements is determined by the value of 'n' which can be 'b' for bytes, 'w' for words or 'l' for long words.

.data1
Equivalent to dc.b

.data2
Equivalent to dc.w

.data4
Equivalent to dc.l

.ds .n m

Define unitialised space. The size of the data elements is defined by 'n' which can be b for bytes, s for words or l for longs. The number of elements of this type is defined by m.

.define
Make the name globally visible outside this module (ie. and XDEF).

.end

.equ

.even
Ensure next instruction is on an even memory address.

.extern
Assume the name is an external name in another module (ie. an XREF).

.globl
Make the name globally visible (ie. an XDEF).

.org

.rom

.sect section
Start the specified section

.space n
Equivalent to ds.b n

.text
Start the TEXT section

Comments can be included by preceding them with either the semi-colon or the exclamation mark sysmbols. If they start in column 1, then asterisk is also accepted.

Hex numbers can be written in either of the following forms:

$00 Normal assembler style

0x00 C style

KNOWN BUGS

The assembler will (now) accept operands of the form label-label. However incorrect code will be generated if both operands are not in the same source file, and both in the same segement. No warning or error message is given.

1. This is due to the fact that this capability was added as a "quick hack" for use by C68 rather than by humans. If anyone does the work to make the support more generic and remove the above restrictions, then please pass the results to D.J.Walker.

CHANGE HISTORY

The following is the change history of this document (not AS68 itself). It is intended to help users who are upgrading to identify the changes that have occurred.

01 Jun 94 DJW
A section added on the fact that label-label constructs are now supported in a limited fashion.

Make

NAME

make - maintain, update and generate groups of programs

SYNOPSIS

make [-f makefile] [-acdeiknpqrst] [target_list]

DESCRIPTION

Make is a utility designed to formalise the relationships between the files that make up a large computer program; a suite of programs; a list of repetitive tasks and provide an easy way of keeping them up to date.

Make may be invoked with various arguments. These are:-

-f makefile
This option, followed by the name of a file, changes the default makefile read from just 'makefile' in the current directory to the name given. Eg.

EX MAKE;'-f arcmake'

would use arcmake as the makefile.

-a
Try to guess at undefined ambiguous macros (such as **$*** and **$<**).

-c
This option changes the line continuation character (by default this is

This option changes the line continuation character (by default this is '\' backslash). It should be followed by the new line continuation character, eg.

-c+ would use +.

This will be overridden if the directive .LINECONT is used in the makefile.

-d
Print debugging information regarding the analysis of a 'makefile'.

-e
Environment variables will override assignments within makefiles.

-i
This performs the same function as the .IGNORE directive. It allows make to continue on error.

-k
Abandon work on the current entry if it fails, but continue with any other targets that are not dependent on the failed target.

-n
Stops **make** from executing any of the commands it would run to put a target up to date. It just prints the commands it would have run. This is useful for testing if a makefile will run the commands you expect it to.

-p
Print all macros and targets after analysing the makefile.

-q
Interactively question the 'up-to-date'ness of targets.

-r
Do not use built in rules.

-s
This has the same effect as the .SILENT directive, it stops make from printing out the commands it is executing before it runs them.

-t
Touch the targets (causing them to be updated) rather than running the usual commands to update them.

Any other arguments to make are taken as targets to make. If no targets are defined, then make will use the first target defined in the makefile. Therefore if a makefile has no list of targets then just running

EX make;' [option list] '

will cause it to make the first target it finds in the makefile. However, if another target is provided on the command line then make will make that target or targets instead of the first target.

Make updates a target only if its dependencies are newer than the target. Missing files are always deemed to be outdated.

CREATING THE MAKEFILE

For example, suppose you are writing a database program in C. The program consists of several source files, ending in **_c** , several header files that the C files include, ending in **_h** . When the **_c** files are compiled they produce object files ending in **_o** . These files are then linked with a library or set of libraries ending in **_a** to produce the final database program. Now, imagine you have altered one of the header (**_h**) files. Some of the **_c** files (the ones that include the changed header file) will need to be recompiled and the entire program re-linked to create the database. You could just keep a note of which files depend on which header files and issue the commands to re-compile them and re-link manually, but make provides a way of re-creating the entire program using a single command.

In order for make to issue the commands to re-compile the files it must know which files depend on which. It reads this dependency information from a special file that the user creates called a makefile. When make starts up (by typing :

EX make

in the simplest case) it reads a file called makefile in the current data directory. This file must contain the list of files belonging in this particular project and their dependency files, plus the commands neccesary to re-create them. Next make looks at the dates of the target files (the files we need to keep up to date) and compares them with the dates of the dependency files (the files that the target files depend on). If any target file is older than any of its dependency files then make issues the commands to re-make the target file. As an example:

Suppose we have a program called sed. This consists of two C source files sedcomp_c and sedexec_c. Both source files #include a file sed_h. The file we want to keep up to date is sed, so this is the first target we include

we want to keep up to date is sed, so this is the first target we include in the makefile, along with the command needed to re-create it from its object files (sedcomp_o and sedexec_o)

```
sed : sedcomp_o sedexec_o
  ld -o flp1_sed sedcomp_o sedexec_o
```

The first part of the line (sed :) tells make that sed is a target, and that it depends on the files after the colon (sedcomp_o and sedexec_o). The next line is the command needed to re-create sed, assuming that the file sed_link is on flp1_ and we wish to create sed on flp1_ also. For the purposes of this example we will assume that the default data directory is on flp1_. Note that the line(s) containing the command(s) to be executed must NOT start in column 1.

The next lines in the makefile are secondary targets (they are listed after the first one).

```
sedexec_o : sedexec_c
  cc -c sedexec_c
sedcomp_o : sedcomp_c
  cc -c sedcomp_c
```

Similarly, the first two lines state that the files sedexec_o and sedcomp_o depend on their respective c files, and that to re-create the _o files from the _c files the compiler cc needs to be run.

Finally the fact that both sedcomp_o and sedexec_o depend on sed_h needs to be stated. This is done by the lines:

```
sedcomp_o : sed_h
sedexec_o : sed_h
```

Note that for this dependency no rule to re-create is given, so the previous rule to recreate _o from _c is used when make is running. This is simply a way of stating dependency.

Thus our complete makefile is:-

```
#
# A test makefile
#
sed : sedcomp_o sedexec_o
  ld -o ram1_sed sedcomp_o sedexec_o

#
# Now the _o dependencies
#
sedcomp_o : sedcomp_c
  cc -c sedcomp_c
sedexec_o : sedexec_c
  cc -c sedexec_c

#
# Finally the _h dependencies
#
sedcomp_o : sed_h
sedexec_o : sed_h
```

Note that **#** is used as a command specifier and white space is freely used. There must, however, be a space between the filenames and colons. The command line to run does not have to be on a separate line from the dependency definition, but can insteaad be placed after it to save space. In this case there is a semicolon in between the dependecy abd command. Our dependency

```
sedcomp_o : sedcomp_c
  cc -c sedcomp_c
```

could have been written thus:

```
sedcomp_o : sedcomp_c ; cc -c sedcomp
```

It would be very tedious however, to have to specify all dependencies so explicitly when working on a big project, so make has some intelligence and some short cuts built in. For instance, make knows that to re-create a _o file you can compile the _c file with the same name and that such _o files depend on their _c files. Also make has a macro facility which makes using lists of filenames much easier. For instance, our previous makefile cound be re-written as:

```
#
# Better makefile
#
CFLAGS = -c -h
```

```
OBJECTS = sedcomp_o sedexec_o

sed: $(OBJECTS)
  ld -o flp1_sed $(OBJECTS)

$(OBJECTS) : sed_h
```

This is much shorter (but a lot less clearer). The changed structure of the make file is as follows. The line

```
CFLAGS = -c -h
```

is a macro that tells make to use these particular flags to pass to the c compiler cc when it re-makes a **_o** file from a **_c** file. The actual command

```
CC -c -h <filename>
```

is now not included. This is because make understands how to create a **_o** file from a **_c** file. It does not, however, know how to turn an assembly language file ending in **_asm** into a **_o** object file - it can be taught how to do this but more of this later. The second line is a macro that introduces a convenient shorthand for referring to the object files that sed is composed of. The line

```
OBJECTS = sedcomp_o sedexec_o
```

means that whenever the characters **$(OBJECTS)** are encountered, they are to be replaced with the characters **sedcomp_o sedexec_o** . The macro name must be preceded by a **$** and surrounded with parentheses to make it easy for make to decide this is a macro, not a file name. So the line

```
sed : $(OBJECTS)
```

expands into

```
sed : sedcomp_o sedexec_o
```

just as we used in our first makefile. Likewise the line

```
$(OBJECTS) : sed_h
```

expands to say that both _o files are dependent on the sed_h file. More than one file can be placed before and after the colons; this just says that all the files before the colon depend on all the files after.

There are special macros already built into make. These are:

***@, $?, $*, $<, *>, $C, $P, $D**

and these will be explained later. Macros and command lines may be more than one line long: to get a long macro (for instance a list of object files file1_o ... file10_o) may be split over a number of lines, so long as each line except the last ends in a line continuation character (\ is the default). Eg:

```
OBJS = file1_o file2_o file3_o \
  file4_o file5_o file6_o file7_o \
  file8_o file9_o file10_o
```

The macro $(OBJS) will expand into the entire list of files.

ENVIRONMENT

The environment is read by **make** . All variables are assumed to be macro definitions and are processed as such. The environment variables are processed before any makefile, but after processing **make** 's default rules. Thus environment variables override default rules, and these in turn can be superseded by assignments in makefiles. If the **-e** option is used with **make** , then environment variables also override assignments within makefiles.

The **MAKEFLAGS** environment variable (if present) is processed by **make** as containing any legal input option (except **-f** and **-p**) defined for the command line. Further, upon invocation, **make** "invents" the variable if it is not in the environment, puts the current options into it, and passes it on to the invocations of commands. Thus **MAKEFLAGS** always contains the current input options. This feature can be particularly useful for "super-makes" whereby make invokes further instances of itself.

IMPLICIT RULES

As supplied make has default rules that allow it to compile and link C programs. Make understands that to re-create **_o** files from **_c** files the command

```
$(CC) -c $(CFLAGS) $<
```

is executed. The macro **$(CC)** is pre-defined to be **cc** , the macro **$(CFLAGS)** is pre-defined to be a null string, and **$<** expands into the filename of the target being made. This allows you to substitute any command with its own flags for the cc command usually used, just by re-defining the **$(CC)** macro. For instance if you bought a better C compiler called wombat_c for the sake of argument, you could force it to be run instead of the ordinary cc by typing

```
CC = wombat_c
```

at the start of your makefile. Likewise, any different flags you may use can be used by re-defining **CFLAGS** . The complete list of default rules built into make is given at the end of this document.

To enable make understand how to create files from another extension an implicit rule can be defined. This is of the form :

```
_(target extension)_(source extension ) :
  <commands to be run>
```

As an example of this the current definition to turn _c files to _o files is

```
_c_o :
  $(CC) -c $(CFLAG) $<
```

An implicit rule must start with an underscore, followed by the extension type of the dependent files, followed by another underscore and the extension type of the target files, ending with a colon. The next line contains the command used to re-create the target file.

As a more useful example, the GST macro assembler assembles files ending in _asm and creates files ending in _rel. These are ordinary object files however, so we would rather that they had an extension _o to match the files produced by the C compiler. To assemble a file wombat_asm in the current data directory of flp1_ to produce a file wombat_o (also in flp1_) you would run the command from SuperBasic

```
EX DEV_MAC;'flp1_wombat_asm -NOLIST -NOSYM -BIN flp1_wombat_o'
```

(**-NOLIST** and **-NOSYM** stops the normally verbose output of the assembler). To turn this into an implicit rule to assemble any file with an extension of _asm into a file of the same name ending in _o in the current data directory the following lines would be added to the makefile:

```
_asm_o :
  DEV_MAC $C$*_asm -NOLIST -NOSYM -BIN $C$*_o
```

INTERNAL MACROS

There are number of internal macros that are useful for writing rules for building targets. They are:

$C

Current data directory

Useful when calling programs that don't recognise the toolkit 2 directory defaults.

$P

Current program directory

as above.

$D

Current destination directory

as above.

$*

Current make target with no extension

Eg. if the implicit rule being used is _c_o and the rule is used to re-create test_o then $* expands into test.

$@

Complete filename of current target being made.

$<

The names of the files that caused the target to be remade

Eg. if the rule _asm_o is used to re-create test_o then $< expands into test_asm

$?

The names of the dependency files that are out of date.

$>

The name of the source file out of date.

To include a $ character in the makefile use $$.

DIRECTIVES

The makefile can also include some directives that tell how to behave while it is running. These are:

.SILENT:
This causes make not to write out the commands it it executing to re-make a file. The default behavior is that it writes out a command just before running it (note that this does not stop output from the commands that make runs however). The same can be achieved on a command by command basis by prefixing a command with a @ character. Eg.

```
@cc $(OBJ)
```

would run cc but not print out the command line before executing.

.LINECONT : <character>
This changes the default line continuation character for make. Normally this is a \, which allows commands and macros to extend over several lines. Eg.

```
.LINECONT +
```

will change it to be a plus character.

.IGNORE:
Including this directive causes make to continue trying to make files if any command returns with a non-zero error code. Normally make will terminate if any of the commands it runs return a non-zero error code; this causes it to soldier on regardless. (Useful if you are redirecting the output of make to a file and just want to read about the errors when you return from coffee !). The same directive can be turned on, on a command by command basis, by prefixing the command with - Eg.

```
-cc $(OBJ)
```

would run cc but ignore any error code it returned.

.SUFFIXES : <list of file extensions>
This tells make in what order it should make files using implicit rules. Normally make just re-creates files with the required extension in the order that they were declared in the makefile Eg. declaring

```
_asm_rel : <command list>
_c_o : <command list>
```

will cause _rel files to be re-created before _o files, however, declaring

```
.SUFFIXES : _o _rel
```

would cause the reverse to be true.

CASE INDEPENDENCE

This version of **make** conforms to the standard Unix standard of all names being case dependent within the Makefile. It is not necessary, however, that the actual files held on the QDOS media have their filenames in the same case as in the Makefile.

FINAL NOTE

The correct operation of make depends entirely on the time stamps of files being correct. So if you don't have a battery back up for your QL clock it is ESSENTIAL that you set the time and date every time you power up the machine, otherwise make will make very silly mistakes.

FINAL MAKEFILE

An example makefile for the 'arc' program (public domain, available from the QUANTA library).

```
# Makefile for ARC
  CFLAGS = -c -h

  OBJS = arc_o arcadd_o arccode_o arccvt_o arcdel_o \
    arcdos_o arcext_o arcio_o arclst_o \
    arclzw_o arcmatch_o arcpack_o arcsq_o \
    arcsvc_o arctst_o arcunp_o arcusq_o arcmisc_o

  SRCS = arc_c arcadd_c arccode_c arccvt_c arcdel_c \
    arcdos_c arcext_c arcio_c arclst_c arclzw_c \
    arcmatch_c arcpack_c arcsq_c arcsvc_c \
    arctst_c arcunp_c arcusq_c arcmisc_c
```

```
arc: ${OBJS}
  ld -o $Carc $(OBJS)

${OBJS}: arc_h

arc_h: arcm_h arcs_h
  touch arc_h
```

DEFAULT RULES BUILT INTO MAKE

The complete list of default rules built into the current release of **make** on QDOS is equivalent to a Makefile made up as follows:

```
# Define default directories
  C = current Working directory (as given by getcwd())
  P = current Program directory (as given by getcpd())
  D = current Data directory (as given by getcdd())
# define default program names
  CC = cc
  AS = as68
  LD = ld
  CO = co
  MV = mv
  RM = rm
  YACC = yacc
  ASM = mac
# define default program flags
  CFLAGS =
  ASFLAGS =
  LDFLAGS =
  COFLAGS = -q
  RMFLAGS = -f
  YFLAGS =
  ASMFLAGS = -NOWINDS
# Default rules when using RCS
  _h,v_h :
    ${CO} -p $(COFLAGS) $< >$*.h
  _s,v_s :
    ${CO} -p $(COFLAGS) $< >$*.h
  _x,v_x :
    ${CO} -p ${COFLAGS} $< >$*.x
  _s,v_o :
    ${CO} -p $(COFLAGS) $< >$*.s
    $(CC) -c $(CFLAGS) $*.s
    ${RM} ${RMFLAGS) $*.s
  _x,v_o :
    ${CO} -p $(COFLAGS) $< >$*.x
    $(CC) -c $(CFLAGS) $*.x
    ${RM} ${RMFLAGS) $*.x
  _c,v_o :
    ${CO} -p $(COFLAGS) $< >$*.c
    $(CC) -c $(CFLAGS) $*.c
    ${RM} ${RMFLAGS) $*.c
  _y,v_c :
    ${CO} -p $(COFLAGS) $< >$*.y
    $(YACC) -c $(YFLAGS) $*.y
    ${MV} y.tab.c $@
    ${RM} ${RMFLAGS) $*.y
  _asm,v_rel :
    ${CO} -p ${COFLAGS} $< >$*_asm
    ${CC} $*_asm -BIN $*_rel ${ASMFLAGS}
    ${RM} ${RMFLAGS} $*_asm
# Default rules when not using RCS
  _c_o :
```

```
    ${CC} -c ${CFLAGS} $<
  _c :
    $(CC) $(CFLAGS) $(LDFLAGS) -o $@ $<
  _x_o :
    ${CC} -c ${ASFLAGS} $<
  _s_o :
    $(CC) -c $(ASFLAGS) $<
  _y_c :
    $(YACC) $(YFLAGS) $*<
    ${MV} y.tab.c $@
  _asm_rel :
    ${CC} $*_asm -BIN $*_rel ${ASMFLAGS}
  .SUFFIXES : _rel _o _c _x _s _asm _h _y \
    _c,v _x,v _s,v _asm,v _h,v _y,v
```

C68 Linker

NAME

ld - Link files to produce program files or RLL libraries.

SYNOPSIS

ld [options] [cfile_o] [-llibrary]'

DESCRIPTION

The **ld** linker is used to produce binary code that is ready to run. This code can be any of:

- An **EXEC** able program that can be run directly.
- A RLL (Runtime Link Library) for use with the RLL system.
- Pure binary code that is loaded via **RESPR** commands.

After the files comprising a C program have been compiled into object (SROFF) files they must be linked, together with selected library routines, to produce an executable program. This is done by the program **LD** , which replaces the program **LINK** that is the traditional linker that is commonly used on the QL.

LD is much more UNIX like in use than **LINK** . This is makes it more consistent with the rest of the C68 system which has its origins on UNIX based system. In addition, the **LD** linker is far more efficient than the original **LINK** program in that it runs about 3 times faster, and also the resulting program is smaller. It is also possible to run **LD** in a compatibility mode where it can produce output in the same format as the original **LINK** program.

The different linkers have used different standards for the way that they store relocation information in the output file. To allow maximum flexibility, the **ld** version 2.xx series has the ability to produce output that uses a variety of different formats:

- **ld** version 2.xx format that was introduced to support the RLL system. This format (which is the default) allows more information to be stored in the target file than earlier formats catered for.
- **ld** version 1.xx. This is the format that was used by all versions of the c68 system prior to release 4.25.
- GST **LINK** format. This is the format produced by the traditional GST linker, and the more recent Quanta **QLINK** variant.

If either of these last two formats are required, then the appropriate command line option must be used. **OPTIONS**

The following command line options are available for use with the **ld** linker.

-f n

The output format required. The values available for 'n' are:

0 GST LINK or Quanta QLINK format

1 LD version 1.xx format

2 LD version 2.xx format

You can also add any combination of the following values although many of them will not make sense unless using LD v2.xx format. For a more

detailed discussion of the implications of these settings, see the section concerning the start of the UDATA area later in this document.

4 Set the UDATA area to NOT reuse the relocation area. If you are using GST format, then you would normally use this option as well.

8 Set the UDATA area to NOT reuse the BSS XREF area. This only makes sense if you are either using RLL libraries, or alternatively have used the option (**-z xref**) to include external references.

16 Set the UDATA area to NOT reuse the BSS RLIB area. This only makes sense if you are linking with RLL libraries.

32 Set the UDATA area to NOT re-use the BSS XDEF area. This is the normal default for a RLL, but not for other target types.

64 Set the UDATA area to reuse the BSS XDEF area. This is the normal default for everything except a RLL.

128 Set the UDATA area to reuse the BSS Header area. Normally if any the BSS XDEF area is present then this would not be re-used.

Default: **-f2**

-L library path

This option allows the order of searching of library files to be changed. The order of searching for libraries is:

- Any library directories specified using the **-L** option. If multiple **-L** options are specified, then these are searched in the order they are specified on the command line.
- The **LIB** sub-directory of the current program (**PROG_USE**) device. For example, if the default program directory is **flp1_** then libraries would be expected to be found in **flp1_lib_** .
- The current data (**DATA_USE**) directory.

If a specified library cannot be found in any of these locations, then an error message will be output.

-l libname

Search a statically linked library looking for any modules in the library that will satisfy any of the currently outstanding external references. If any such modules are found then add them to the output file.

Library names are normally specified on the command line after all the object files. This is, however, not mandatory. The libraries and object files are processed in the order specified on the command line. Note that only a single pass of a library is done so that including a library too early in the command line may cause routines to not be included that you might want.

By convention library names are given in a specially compact format. The names for libraries are made up by adding a prefix of **'lib'** and an extension of **'_a'** to the name specified on the command line. For example the main C library is called **libc_a** and the maths library is called **libm_a**.

Thus to link with the math library, you would use **-lm** , to link with the standard C library **-lc** . A full library pathname is never given in this parameter, but you can specify a search path (using the **-L** option). So, if you wanted to link with a private library called **libmine_a** , which was in a directory **flp1_mylib_** then the two options you would need on your command line are :

-L **flp1_mylib_** <object files .... > **-lmine**

where:

flp1_mylib_ = library path

-lmine = library to search

It is not necessary to specify the default C library (using **-lc)** as if there are any outstanding references it is always searched as the last library (although you would do no damage if you specified it explicitly). This library includes most of the routines described in the C library description, along with many other run time routines that need to be included.

-m listing_options

This causes a listing file to be produced for the program. The name of the file produced is always **<output_file_name>_map** so if no output name has been specified then the map file will be called **a_out_map** . If the output file had been specified using **-otest** then the listing file would be called **test_map** .

The amount of detail included in the map file will depend on the options supplied. There must be at least one option, but you can specify more than one by simply concatenating them together (e.g. **-mls**). The options available are:

a A symbol listing will be produced in address order.

l A library listing will be produced. This will indicate each library that is searched and where it is located. If the **m** option is also used, then the library detail lines will be inserted at the appropriate points in the module listing.

m A module listing will be produced showing the details of each module that is included in the link. This will be in the order in which the modules are included.

s A symbol listing will be produced giving symbols in alphabetical order, and showing for each symbol where it is defined.

x A cross-reference listing will be produced showing for each symbol where it has been called from. This option will also imply the **'l'** option as that information is also listed for each symbol.

Note, however, this will miss any cases where the reference to a symbol is in the same module that it is defined in as this is handled purely at the assembler stage.

There is also a listing summary produced that shows the link statistics. This is inserted after the library/module listings and before the more detailed symbol listings.

Note that if the **-v** option is used and there is no listing file being produced, then this summary information is written instead to the console.

-o outputfile
This allows the name of the output file to be specified. For example using a **-otest** would cause a program called **test** to be produced.

Default: **-oa_out**

-R name[**/** version]
This indicates that LD is to build a RLL rather than a normal program. The 'name' part of the parameter is the name that will be given to the RLL thing. This has a maximum length of 16 characters - and more than this will be ignored.

The (optional) version part is the version number (which is of the form **m.nn** where **m** is the major version number and **nn** is the minor version number. In practise **LD** will do no validation on this version number beyond checking its length to be 4 characters and will simply copy the value supplied into the RLL header. If the version number information is not supplied, then a value of "1.00" will be used.

The use of this parameter will also modify the default behaviour of **LD** as follows:

- The default start-up module is changed to be **rll_o** as this is the one normally needed when building a RLL.
- The output program will have any required UDATA space included in the file to simplify the loading process for the RLL Runtime Link Manager (RLM).

-r libname
Search a RLL library. A RLL library is one that is dynamically linked to a program at runtime. Although LD does nto add it to the output file it does search it to determine which external references would be satisfied by including this library. The same locations will be searched as are specified earlier for the **-l** option.

The name of a RLL library is made in a similar way to that for statically linked libraries except that the extension is **_rll** . For example the maths library would be specified by using **-rm** which would cause LD to look for a RLL library called **libm_rll** .

RLL libraries can also have what are called 'RLL stub libraries'. These are small statically linked libraries that need to be linked in to help interface to a RLL. These have the same name as the associated RLL, but with an extension of **_rls** . Therefore the stub version fo the maths library specified using **-lm** would be **libm_rls** . LD will look for such a stub library any time a RLL is specified, and if there is one it will statically link it in before the RLL library file. However stub libraries are optional, and no error will be given if no stub library is found.

Note that if a RLL library is linked in, it is necessary for this library to also be present at runtime for the program to run successfully. For more information on the RLL system see the RLL_DOC file.

The other point to note is that no RLLs are linked in by default, not even the **libc_rll** file. This is to ensure that unless the user explicitly asks for RLLs to be used the default action is to do static linking as in earlier releases of the **ld** linker.

-s startup_file

This option specifies a different startup file from the standard one. Using the special format of **-s-** means that no start-up module is to be used.

The startup file is the code that does all the run time relocation that allows C programs to run correctly where they have been loaded in the QL. Unless you have written your own startup file that does runtime relocation correctly it is probably best to leave this option alone.

If this parameter is ommitted, then the startup modules that are used will be:

crt_o for standard programs

rll_o for RLL libraries

The startup file is searched for using the same paths as are used when searching for libraries (as described under the **-l** option). Examples on when you might want to used different values are:

-screspr_o if you are using C to build code that is going to be loaded as a resident extension rather than used as an **EXEC** able program. For more details see the **C68QDOS_doc** documentation file.

-s- if you are going to use LD as a replacment for the **LINK** linker and you are not writing C68 compiled code.

-V

Output a message giving information about the version of **LD** being used. This message will be sent to the listing file if one is to be produced, and otherwise to the console.

-v

This means run in verbose mode. It also implies the **-V** parameter. This will always cause the program version to be output a link summary to always be produced. If no listing file is being output these will be sent to the console, otherwsie to the listing file.

-z defs

Force a fatal error if undefined symbols remain at the end of the link process. This is the default action.

-z nodefs

Allow undefined symbols. This might be used if you are developing a program and you know the symbols in question will not be used. It can also be used when building a RLL to allow an undefined RLL to be linked in dynamically at run-time. However as any attempt by a program to actually use an undefined symbol is likely to cause a system crash this option should be used with extreme caution.

-z udata

Store any **UDATA** space as part of the file data rather than simply storing the size required as part of the information in the file header.

This is the default if building a RLL.

You are recommended to use this parameter when building code that is to be loaded via RESPR or LRESPR instructions.

-z xref

Include a list of any unsatisfied external references in the program, and the details of the RLL that will satisify them. This is very similar to the **-nodefs** option mentioned above, except that this option expects you to have used the **-r** option to specify the RLL that will be used to satisfy the reference at runtime.

This is actually the default when building the **-r** parameter is used to link in a RLL.

-z xdef

Include a list of externally visible definitions that are contained in this program or RLL.

This is the default when you have used the **-R** parameter to specify that you are building a RLL.

The following options will also be recognised by **LD** for compatibility with earlier versions, but they will simply be ignored and have no effect. This is because the relevant areas are now allocated dynamically.

-bufl size **[K]**

This allows the user to specify the buffer size in reading the object files and library files. The default buffer size is 8K. This is quite small, but as ld does all its work in memory, its requirements for memory space are quite fierce. Unless linking is unacceptable slow or you have lots of memory to spare it is probable best to leave this option alone (except to decrease it). Eg. -buf32K would allow a 32K buffer for reading library and program files.

-bufp size **[K]**

This allows the user to specify the size of buffer to hold the complete image of the program that is being linked. Normally this is set to 50K

image of the program that is being linked. Normally this is set to 50K which is enough for most small programs. If you want to link a very large program then use a larger value here. Alternatively if you know your program is very small then you could sacrifice program buffer space for library buffer space.

Following the options the object files to link are specified. These may be any valid QL filename, with the directory extensions provided by a C program (eg. .._.._test_o, test1_o, etc.). No wildcards are allowed in the ld command line as the order of files linked is important in ld, and this could not be guaranteed if wildcards were used.

The linker produces files that are smaller than the equivalent produced by the GST 'link'. The program files produced set the job data size field in the QDOS files header. THIS MUST NOT BE MADE SMALLER by any toolkit routine as if this is done the programs produced will FAIL to relocate properly at runtime and crash the QL ! YOU HAVE BEEN WARNED ! Making it bigger is a waste of space as this data area is only used during program initialisation, and is not used for the runtime stack or heap.

ENVIRONMENT VARIABLES

The LD linker will make use of the Environment Variables specified below if they are set. These environment variables are processed before the command line options, so in the event of any clash, the command line parameters will take precedence.

LIB_DIR
This can be used to specify the default program directory.

LD_OPTS
Any parameters taht would be valid on the **LD** command line can be set in this environment variables.

START OF UDATA AREA

This section is only relevant to those who are trying to make advanced use of **ld** . In normal use one will not need to use (or even understand) the contents of this section.

It revolves around the fact that normally the information held in the **BSS** section of a program or RLL is often only required at initial load time. After that it is no longer required. It would therefore save on memory if this space could be re-used for other purposes. Most programs also contain a **UDATA** section that is used to hold uninitialised data. If you can re-use this space after the initialisation phase is finished then the program will need less memory to run.

Another aspect to consider is whether any space for the **UDATA** area should actually be included in the data stored as part of the program file header. Doing so reduces the size of the file stored on disk, but means that the loading process needs to take account of this.

The algorithm used by default with **ld** is as follows:

- For a standard statically linked program, the **UDATA** area starts at the same address as the start of the BSS Header. The file header is used to store any additional space that needs to be allocated to complete the **UDATA** area. The BSS area will include the following sections:
- For a standard statically linked DLL that does not call another RLL, the **UDATA** area starts after the end of the BSS XDEF area. Any additional space required for **UDATA** is included as part of the data of the program file.

This default behaviour can be modified by use of the various **-f** and **-z** runtime options.

CHANGE HISTORY

This section details the major changes that have been made to this document. It is intended primarily for those who are upgrading their release of C68 to help them identify when and where new information has been included.

31 Dec 93 DJW — Added section on Environment Variables that LD will now recognised.

28 Mar 96 DJW — Updated to reflect the new options that are now available for use with **ld** version 2.

C68 Menu System

C68Menu v4.0

1. INTRODUCTION

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

1. Select from a directory listing rather than type in a filename
2. Forget how to start Editors, Compilers, Linkers etc
3. Forget compiler and linker switches like -v -ms etc
4. Add libraries without remembering how to do so, or their names
5. Use make files/alter make files but never have to edit one
6. 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 irrelevent 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 labelled EDIT, MAKE, EXEC and OPTIONS. Selecting these will initiate the appropriate action. A fifth box labelled 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 progam.

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

977740727277979727470400

The top line indicates to what windows the lower lines refer. <FORM> refers

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

C68 environment on QDOS and SMS

Release 4.25

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

_minneed
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_1, 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:

1. The character ***** is used to represent a string of any characters, and of indeterminate length.
2. The character **?** is used to represent a single character of any value.
3. 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').
4. 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 Qubata **QLINK** program which is just a

the standard GST **LINK** program (or the Qubata **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.

CHANGE HISTORY

This is a short summary of the changes that have been made to this document. The intention is to make it easy for users who are upgrading to find any new information.

02 Oct 93 DJW	Changed the statements that show how global vectors are set to include parameter types as required by C68 Release 4.00 onwards.
10 Nov 93 DJW	Removed part about possible conflict with assembler reserved words. Cannot now happen as all external C symbols now start with an underscore character.
31 Dec 93 DJW	Removed section on Winchester disk support as no longer really relevant.
	Changed statements for initialising global function vectors as parameter types no longer required (assuming QDOS_H has been included (i.e. undid change of Oct 93).
25 Apr 94 DJW	Minor changes and corrections for the 4.13 release.
03 Sep 94 DJW	Added paragraph on using fflush() for mixing level 2 I/O with levels 1 or 0 as an alternative to running channe; unbuffered.
20 Jan 95 DJW	Added description of the **_Cinit**, **_Readkbd** and **_Open** vectors.
10 Aug 95 DJW	Added comments to section on default directories and simple device clashes
28 Sep 95 DJW	Added description of **_endtimeout** global variable to section on termination message.
	Added short section on signals.
28 Apr 96 DJW	Moved section on using GST LINK program to this document as being a better lcoation than the documentation of the ld program.

Environment Variables

INTRODUCTION

This document discusses the topic of Environment variables in C68. It covers how they are set up, how they are accessed, and why you might want to use them in the first place.

WHY USE ENVIRONMENT VARIABLES

Environment Variables are an idea that has been adopted from the UNIX operating System. Environment variables are basically global variables that use to control certain aspects of how your system will run. Typically you would set up standard values during the loading of your system, although you may modify some of them at a later date.

In QDOS terms, this means that you can set up information in your BOOT file that can be later interrogated by other programs running. This can make it much easier to centralise the control of your system rather than having to tell each program separately as you run it.

An important feature of Environment Variables is that they are NOT cleared if you load a new SuperBasic program. Thus they remain set for the duration of your session (or at least until you reset your machine).

SUPERBASIC INTERFACE

The SuperBasic interface is provided as four extensions to the SuperBasic interpreter, the code for which in contained within the ENV_BIN file provided with the standard C68 distribution. This file should be **LRESPR** 'ed if you wish to make use of the facilities that it offers.

The four extensions provided are as follows:-

SETENV	A procedure that allows you to set the value of an environment variable
ENV_LIST	A procedure to list the settings of your current environment variables to a channel (usually the screen)
ENV_DEL	A procedure to delete an existing environment variable
GETENV$	A function that returns the value of a specific environment variable

GETENV$ variable.

USING THE SUPERBASIC INTERFACE

The first thing that you are likely to want to do is to set the value of one or more environment variables. This is done by using SuperBasic statements of the form:

```
SETENV "NAME=VALUE"
```

This sets up an environment variable NAME, and gives it the value VALUE. For instance, the command

```
SETENV "TMP=ram1_"
```

gives the environment variable TMP the value of ram1_. It is worth noting that the VALUE part may be null, ie

```
SETENV "TMP="
```

will assign a null string to the variable TMP. This should be contrasted with the putenv() usage, below as used in the C environment.

Subsequently setting another value for the same variable causes the previous definition to be overwritten. Most users will set values at startup from within their boot files.

You can obtain a listing of the current settings of all your Environment Variables at any time by using a SuperBasic command of the form:

```
ENV_LIST
```

or

```
ENV_LIST #n
```

This causes all currently defined environment variables to be written to the channel specified, or #1 by default.

If you decide that you wish to completely remove an Environment Variable, then this can be done by using a SuperBasic command of the form:

```
ENV_DEL "NAME "
```

This procedure will remove an environment variable definition completely. In many cases this will be equivalent to using the SETENV command to give a variable a null value. They are not quite the same as at the C level one will result in the getenv() call returning NULL, and the other will result in it returning a pointer to a zero length string.

If you want to interrogate the value of a particular Environment Variable from within a SuperBasic program, then this can be done by using the function:

```
GETENV$("NAME" )
```

This allows the environment variables to be manipulated from SuperBasic. The GETENV$ function will return the value of and environment variable, or a null string if not found. Hence, after:

```
a$="test=debug level 1"
SETENV a$
```

PRINT GETENV$("test") would print the string "debug level 1"

but

PRINT GETENV$("TEST") would print a null string (since Environment variable names are case sensitive).

and the sequence

```
b$ = GETENV$("TMP")
IF b$ = "" THEN
  b$ = "RAM1_"
  SETENV "TMP=" & b$
ENDIF
SETENV "TMP_FILE=" & b$ & "WORK_TMP"
```

will ensure that the TMP variable is set, and that another environmental variable, TMP_FILE, contains a reference to a file WORK_TMP on the same device.

C68 C LEVEL INTERFACE

The Environment Variables capability interface is built into all C68 programs that are compiled with Release 2.01 or later of C68.

The routine main() is now passed an additional parameter at program startup that points to the environment variables (if any) that are currently set up. The syntax of main() is now therefore:

```
int main (int argc, char * argv[], char * argp[])
{
 ...
```

where the **argp** variable is an array of string pointers. Each entry points to a single NULL terminated string which is in the form:

```
STRING=VALUE
```

and the list is terminated by a NULL pointer.

A user program has a number of library routines available to manipulate

environment variables:

- the routine **getenv()** can be used to obtain the value of a specific named Environment Variable. A NULL pointer is returned if the specified Environment Variable does not exist. i.e.
  getenv("TEMP");
  would return a NULL if the TEMP Environment variable did not exists, and a pointer to its value if it did. Please note that the returned value may also be a zero-length string for an Environment Variable that does exist, but that has not been assigned a value.
- the routine **putenv()** can be used to give a value to an Environment variable. This will change the value if the named Environment Variable already exists, or create an new value if does not. Therefore a call of the form:
  putenv("TEMP=ram1_");
  would set the environment variable TEMP to a value of ram1_.
  If you want to remove an environment variable completely then call it assigning a zero length value. i.e.
  putenv("NAME=");
  would delete the environement variable NAME (This is a slightly non-standard treatment).

INHERITING ENVIRONMENT VARIABLES

If a number of c68 compiled jobs are chained together then the environment variables of each slave job are inherited from the owner. The master job will obtain its values from the settings in SuperBasic.

This inheritance factor is important as it means that if a library call is used to change one of the environment variables, then this is remembered and passed on **without** changing the setting that is current at the SuperBasic level.

STANDARD ENVIRONMENT VARIABLES

A number of standard Environment Variables are set up automatically by the C68 system. These are

PROG_USE

DATA_USE

SPL_USE

They contain the values for the Program, Data and Destination directories respectively. If the program has been started from Superbasic and these Environment Variables have not been explicitly set at the SuperBasic level, then these Environment Variables will have the same values as you have set up using the Toolkit 2 commands PROG_USE, DATA_USE and SPL_USE.

TIP: If you want your C programs to use a different default directories to those used by Superbasic programs, then you can set any one of the DATA_USE, PROG_USE or SPL_USE environment variablesexplicitly at the SuperBasic levle.

WHAT PROGRAMS USE ENVIRONMENT VARIABLES

To determine whether a program will use any environment variables, it is necessary to look at the documentation for that program.

Examples of programs that are supplied as part of the C68 system that DO make use of Environment Variables are the **CC**, **C68** and **MAKE** programs - refer to the documentation on each program for the details.

AUTHOR(s)

Original version - Dave Nash.

C code modification - Dave Walker.

SuperBasic interface extension - Dave Woodman.

CHANGE HISTORY

This is a short summary of the changes that have been made to this document. The intention is to make it easy for users who are upgrading to find any new information.

25 Apr 94 DJW Minor changes and corrections for the 4.13 release.

QDOS/SMS Signal Handling Extension

1. INTRODUCTION

This document describes the implementation of the SIGNAL device driver for QDOS and SMS. Initially C68 compiled programs have been seen as the main users of this facility, but the implementation is actually language independent so that it can be used from any language [although processing signals by good old SB interpreter could pose a problem].

This document assumes that the reader has some basic information

This document is structured in such a way that the general information is given first. Subsequent sections get deeper into the way that the SIGNAL system has been implemented. You only need read, therefore, as much of the document as you think is relevant to the use you want to make of the SIGNAL system.

The SIGNAL system was developed by:

Richard Zidlicky
[address and contact details removed at request of author]

If you have any feedback on the SIGNAL ssytem, then it can be provided either directly, or via Dave Walker (Issue co-ordinator of the C68 system).

2. BACKGROUND

In UNIX and similar operating systems signals are used for a number of purposes:

1. signalling and handling soft and hardware errors and exceptions like stack overflow, illegal ops, div by zero, bus and address exceptions
2. notifying processes about a situation needing urgent response, for example communication with sockets or special devices
3. for (asynchronous) communication between processes
4. handling of certain keyboard or timer events like ctl-c(INT) or ctl-s, alarms, or events generated for example by the window manager (SIGWINCHD)
5. telling the process that it exhausted its limit on certain resources like memory, CPU time, number of open files

In all of the above cases the normal execution of the process is interrupted (at any point), its context saved and the handler for the event is called. If the user program has not supplied an explicit handler for a particular signal type, then the system supplied default handler is used instead.

In QDOS and SMS option (a) is already (largely) implemented through MT.TRAPV but, unlike signal handlers, the handler routines are called in supervisor mode and also need to take care of the differing stack frames formats used by different members of the 68xxx family of processors. Sooner or later the signal extension will offer some option to take care of this.

It is my intention to provide an implementation for (b), (c), (d) and (e) in QDOS through the signal extension.

At the moment (c) and (d) is working fine (at least on some machines).

The mechanism provided means that in principle (b) and (e) is no problem either, but you need some special way to generate the events - a job checking for a special keyrow combination or a button or hotkey appear more appropriate in a QDOS enviroment than rewriting console drivers to send signals on ctl-\ etc.

The SIGUTIL program (described later in this document) is a simple exmaple how signals could be interactively generated in QDOS or SMS. A more elegant way of doing this would be to define buttons representing the signals that could be drag & dropped into the application's window.

(e) would work but QDOS does not use this possibility - it is therefore only useful at the moment for resources that are not managed by the operating system.

(b) (and partly also (e)) require that it is possible to send signals from within device driver code or extern interrupt handlers, currently this can be done only from the scheduler loop.

3. INSTALLATION

To load the signal extension, use the instruction

LRESPR sigext_rext

If in addition, you use the pointer interface and you wish interactive signal generation via the SIGUTIL program then use a command of the form

ERT HOT_RES(chr$(236),sigutil) : rem alt-F2

Note that if your QL has other than 50 Hz poll loop frequency, you should run the **config** program on **sigext_rext** to adjust it, otherwise alarm() and related calls will be pretty unprecise.

Lightning is known to upset the timing mechanism of the signal extension, _lngOFF apparently cures the problem.

4. SENDING SIGNALS USING SIGUTIL

The SIGUTIL program is a simple interface that allows you to send signals to any program running on the system.

Activating SIGUTIL

The SIGUTIL program is activated by whatever hot-key combination you have chosen (the example above uses ALT-F2).

Sending Signals

- Get the active cursor or mouse pointer into a window of the job you want to send the signal
- press ALT-F2 (or whatever hotkey you assigned to it), SIGUTIL will open a window with a primitive menu and show the jobname and some options.

Now you can simply press ENTER to send this job SIGINT, this is what you will do most of the time, or press some of the other keys for other signals or options.

SIGUTIL will now try to send this signal, if there was an error it will show the error message for a couple of seconds.

Setting uval

'uval' is an extra parameter that can be delivered to the signaled job, SIGUTIL zeroes it by default.

To change this hit '**u** ' in the main menu and edit the value.

Setting piority

SIGUTIL uses the highest possible priority for sending signals by default.

To change it hit '**p** ' in the main menu and adjust with the arrow keys.

```
 WFJ  applies if the job is waiting for another job
 SYSC ... I/O
 SUSP .... suspended
```

The defer flags should always be enabled for c68 programs.

If you do not want to use the programming interface to signals, then you can probably skip the remainder of this document.

5. DIFFERENCES TO UNIX SIGNALS

There are still some, but using standard C library calls you may never notice the difference. While some of the differences are considered usefull features, other may disappear sometime.

Some of the differences :

- Default actions are very different from unix, for example signals can't cause coredumps.
- Some unix flavors allow an additional parameter to be passed with certain signals, I have generalized this behaviour, so that some routines can take an extra 'uval' parameter. This can be used for example to pass extra information about the cause of an exception.
- Receiving signals is controlled by a set of priorities and various flags in addition to the unix control schemes. A job that does not explicitly establish a handler will simply ignore all signals. (ERR_NF being returned to the signaling job)
- Because Unix system calls are often emulated by larger chunks of code in QDOS I have created an 'emulated system call' feature to allow better control of such code.
- Unlike unix, many signal numbers do not yet have any special meaning to QDOS.
  SIGKILL and signal number#0 get special handling in QDOS - 0 can be used to test whether a job exists and has established an signalhandler. (err_bj, err_nf)
  SIGKILL can't be blocked or caught using the '%CSG' handler. (But the default action may still be defined by the job in same cases)
- Sending and receiving signals works through channels, this is only an implementation detail. Sending signals should also work over networks.
- In UNIX pending signals are checked whenever the process exits the kernel. This is not yet done by QDOS, so obviously checksig() must be called explicitly or through sigcleanup() in some situations - currently only when a signal handler is exited through longjmp().

6. USING SIGNALS FROM C

The easiest way to use the SIGNAL extension from within C programs is to use the routines supplied within the standard C library. This library provides routines that emulate the functionality of all the signal handling routines defined by both POSIX and UNIX.

If you use the library routines anywhere in your program, then you do not have to do anything additional to get the signal handling support included into your C program. It will automatically be included from the C library simply because you have used one of the relevant signal handling library

routines.

If you use no signal handling routines anywhere in your program, then by default no signal handling code will be included. In this case the handling of any signals for such a program will be determined by the default handling of signals that is built into the signal extension. If you want to include default signal handling explicitly into your program, despite the fact that you may have used no signal handling related library calls then you can do this by including a call of the form:

```
_Signals_Init = &_SigStart;
```

at the start of your main() function in your program.

Signal Extension not loaded

Another possibility is that your program **is** set up to support signals, but that at runtime the signal extension is not present. The default is that nothing happens, and your program runs as normal, but simply returns an error code to any signal handling functions. What happens here is determined by the behaviour of the routine **_SigNoImp** that is called any time the any attempt is made to access the signal handling sub-system (including initialising it). The protoype for this function if you want to return your own is

```
int _SigNoImp (int signal_number, ...);
```

The library provides a default implementation that will output the message pointed to by the global variable **__SigNoMsg** (unless this is NULL) every time the signal handling sub-system is called (including the initial initialisation) until the message has been output the number of times defined by the **__SigNoCnt** global variable. The routine then returns the QDOS/SMS error code for "not implemented". The default definitions for **__SigNoMsg** and **__SigNoCnt** are

```
short __SigNoCnt = 1;
```

```
char * __SigNoMsg = "*** SIGNAL extension not loaded ***\n";
```

Initialisation of Signal sub-system

If you use the default C start-up code, then you do not need to do anything to get the signal sub-system initialised if you have used . If, however, you have suppressed the C start-up code, then you can still use the signal sub-system if you initialise it explicitly. To do so you need the following line in your program:

```
_Signals_Init();
```

All signal handling functions will then work as normal.

Cautions

There are some restrictions that one needs to be aware of when using signals within your program:

- QDOS/SMS signals can interrupt trap #3 routines, waitfor() and suspend calls. Such calls return ERR_NC as the return code. The C68 calls that mimic UNIX system calls should finally handle this correctly. STDIO level calls such as fread() and fwrite() will normally retry the I/O operation and continue without error.
  Programs that call trap#3 routines, mt_susjb or mt_cjob directly should be aware that these operations can return prematurely.
  If this is a problem you can either set priority for such calls or do a retry by the signal handler. However it is better to write programs so that they do not depend on this.
- There are restrictions on what library routines can be called from within a signal handler. In particular signal handlers should not call any c68 memory management routines (unless you know very well what you are doing) or STDIO routines. Caution should be used when calling other library functions or modifying global variables used by library routines.

Compatibility

I hope this interface does emulate some unix flavors fairly well, but certainly programs that rely on coredumps or any default actions will have problems.

Signal related Library functions

The following POSIX style functions are supported. They are described in more detail in the LIBUNIX_DOC file:

```
alarm()
kill()
raise()
signal()
sigaction()
```

sigaction()
sigaddset()
sigdelset()
sigemptyset()
sigfillset()
sigismember()
siglongjmp()
sigpending()
sigprocmask()
sigsetjmp()
sigsuspend()

The following additional variants on Unix functions are supported. They are also described in more detail in the LIBUNIX_DOC file under the more standard variant:

fraise()
raiseu()
killu()

The following functions that are specific to the C68 implementation of signals are supported. They are described in the LIBC68_doc file

sendsig()
set_timer_event()
sigcleanup()

Some global variables that can be used to alter default behaviour of the handler and some library calls

struct SIG_PRIOR_R _defsigrp default receiving priority
struct SIG_PRIOR_S _defsigsp .. sending ...
struct SIG_PRIOR_S _defsigskp .. for sending SIGKILL

_defsigrp is only looked up during program startup and (notyet) after every sigsuspend() call.

7. TECHNICAL DESCRIPTION

This section covers low level details of the implementation of the signal extension. It would not be necessary for the average user to be aware of any of the details in this section.

QDOS signal interface

The signal extension is implemented as the SIGNAL device driver Opening a channel to this device establishes a signal handler for the job and/or enables sending signals.

Sending signals and controlling timer events is done by sending messages through trap#3 I/O calls. io_fstrg can be used to enquire the **version** of the installed signal extension and some additional information.

C programs (and other that use **%CSG** type handlers) get access to a vectored set of calls to control their handlers and raise signals - see **%CSG** handler interface. C programs can also use the standard Posix defined set of library routines for handling signals.

QDOS signal handlers:

This section describes the differences between %SIG and %CSG handlers. However it is recommended to use the automatic handler initialization, **goto next chapter**.

When initially opening a channel to the SIGNAL device, one can ask for two different modes of operation. The mode required is defined by the type of structure that is passed as a parameter to the open.

%SIG is a very simple, stateless handler intended for use in assembler programs. There is only one handler routine getting signal number and some other parameters passed in registers.

%CSG is a complex handler structure intended to emulate UNIX signals as close as desirable for C68 programs, uses C style parameter passing and provides a vector for some functions

Both structures are defined and partly initialized by the job in its private memory, where they must remain allocated for the lifetime of the handler. A major difference between them is that **%SIG** handlers don't allow you to block signals, while the **%CSG** handlers can do this. Also **%CSG** allow to ignore certain signals so that the job is not at all disrupted when they occur.

C programs would normally use the standard library routines for manipulating signals, and these are mapped onto the **%CSG** handler. There is no technical reason, however, why C programs cannot use any of the handlers directly using trap#3 style calls if they need to implement functionality that would not otherwise be available. If you wish to access the signal structures from within a C program, this can be done by using

#include <sys/signal.h>

The QDOS_SIGH structure corresponds to the **%SIG** structure, while the QDOS_CSIGH structure corresponds to the pretty complex **%CSG** handler.

Assembler programs should probably use the simpler **%SIG** type handler. The layout of the **%SIG** handler is

Offset.size	Value	Description
0.w	$4afc	magic
2.l	'%SIG'	type
6.l	sighnd	pointer to sighandler or SIG_IGN
10.l	stackbot	don't signal if a7 < stackbot
14.l	priority	priority

Here the layout of the **%CSG** handler as defined in sys/signal.h . All arrays have to be **(nsig+1)** ellements. Some features are not yet fully tested or implemented, they are not described here. See the c68 library sources for an example how to initialise this structure.

```
struct QDOS_CSIGH {
unsigned short m1; /* magic, init to 0x4afc */
unsigned long m2; /* type = '%CSG' */
unsigned nsig; /* # of signals to be handled */

void (*** curr) (int) ; /* current table * */
void (*** def) (int) ; /* default table * */
  /* def=NULL is legal since signal extension v0.27 */
  /* thus using default table builtin siegxt instead */
sigset_t **samask;

unsigned short prot_id; /* VOID, init to 0 */
union {unsigned short levl;
  unsigned short **plevl;} hprotlevel; /* VOID, init to0*/

unsigned short stack_id; /* currently must be 0 */
char * stackmin; /* lowstacklimit */

unsigned short pri_id; /* 0= signle value, 1=array */
union {struct SIG_PRIOR_R prio;
  struct SIG_PRIOR_R **pprio; } priority;

struct *statinf; /* pointer to syscall control */
 /* struct, see system call control*/
int (*sigvec)(int, ...); /* vector for sigaction() etc. */

 /* some status info */
unsigned short cprotlevel; /* void */
unsigned short defer; /* never touch !!! */
unsigned short activ; /* never touch !!! */

long **uval; /* pointer to array or NULL */

sigset_t block; /* don't use these */
sigset_t pending;
sigset_t resethandler;
 };
```

Initializing a signal handler for sending/receiving Signals

For receiving signals a job should open a **'*SIGNAL_R'** channel. The code in d3.l can be the addres of one of the structures defined in the last chapter, the prefered method is to use D3.L=0 and setup the handler as described in the next chapter.

To send signals a job can use any channel it or any other job has opened for sending or receiving signals. It may open a send only channel with the name **'*SIGNAL_S'**

The following io.open errors can occur:

err.nf not found. Forgotten to lrespr sigext_rext?

err.iu trying to establish more than one handler for this job

err.bp *signal_' instead of '*signal_r', invalid address in D3

Automatic handler setup

Just send a SIG_HIMSG to the channel opened for receiving channels (see also chapter about messages).

```
/* message struct for initialising handler */
struct SIG_HIMSG {
  unsigned long magic; /* '%MSG' */
  unsigned long len; /* sizeof(struct SIG_HIMSG) */
  unsigned short type; /* M_HINIT */
  unsigned short txi; /* flags, see below */
  unsigned long jobid; /* must match owner id */
```

```
    unsigned long hi_nsigs; /* _NSIG */
    unsigned long hi_stack; /* stack bounds */
  };
```

Currently this flags are implemented:

```
  #define HI_UVAL 0x1 /* alloc array for storing uval */
  #define HI_APRIO 0x2 /* allow individual priority for each signal */
```

After this initialization it is usefull to get the address of the **sigvec** interface routine, see chapter about trap#3 calls

Signal handler routine(s)/ receiving signals

Sending a signal to a job causes this job to interrupt its normal work and jump to the signal handler (sub-)routine, comparable to a hardware interrupt.

'%CSG' handler(the default case):

handler routine receives its parameters on stack, the complete declaration of the handler routine is

```
  void handler(int signo,int pri,int uval, struct ESVB ctxt);
```

'%SIG' handler:

handler routine gets these parameters:

```
 D4.L =  signal number ; upper word reserved, 0
 D5.L =  priority of signal ; any use for this?
 A4.L =  usval ; extra parameter
 A7.L =  points at (struct SIGSVB) ; this may be used to examine the state
         in which the job was interrupted
```

only D4,D5,D6,A4,A5,A7,SR are saved by the system (into struct SIGSVB), the signal handler routine must save all other registers it uses.

In both cases the handler routine is called in user mode.

Exiting from signal handling

The signal handler can be exited locally (RTS, return()) or nonlocally (longjmp()). C programs should use sigcleanup() call if exiting nonlocally, otherwise pending signals might not be processed.

Signals that interrupted a QDOS call

A job should allow such signals (see priority) only if it can handle the resulting conditions. Only system calls with timeout greater 1 or infinite can be interrupted.

If an I/O (trap#3) call is interrupted, it will return AFTER the signal handling (if ever) with D0=err.nc and all other registers as if a normal timeout had occurred.

The signal handling routine may examine the state in which the job was interrupted - but note that struct SIGSVB describes the state as it was when the signal was sent not when it was received. This means eg if SIGSVB.signature indicates an IO call may have been interrupted, this is only true if D0=-1

This is because interrupting the suspended state is done by setting jobheader.stat (acting as timeout) to a small positive value, giving QDOS some chance to complete its task.

System call control

Because many Unix calls have no exact equivalent call in QDOS, they are often translated to one or more QDOS calls by the library. Unfortunately it turned out that without special treatment of such calls it is very difficult or even impossible in some cases to emulate the correct behaviour of such calls with respect to signals.

The best way to get the desired efect would be to make them real system calls, this is not practical in the current implementation for a variety of reasons.

A mix of strategies is used to achieve a good emulation without too much incovenience:

- a special sigvec function is used to enter system calls, this is a 'call through' mechanism. This function also establishes a context to where program control returns in case of a fatal signal.
- in SYSCALL mode all but fatal signals are blocked, the system call itself can check for pending signals.
- a special mode is provided to unblock QDOS calls with infinite or very long timeout

Fatal signals (in this context) are SIGBUS, SIGILL or SIGSEGV, not SIGKILL! If one of these is received, program control returns per longjmp out of the system call function.

Other signals (unless blocked or ignored by user code) are blocked. If the SCTL_EXP mode is used, any pending signal(s) will cause interruptible QDOS

calls that are currently executed or called later while in SCTL_EXP mode to timeout.

As in most situations it makes litle difference whether the signal originated from a raise() or some other cause

This macros are defined for system call management:

```
err=SYSCALL0(flags,&sctl,syscall)
err=SYSCALL1(flags,&sctl,syscall,arg1)
........
err=SYSCALL3(flags,&sctl,syscall,arg1,arg2,arg3)
```

currently no flags, syscall() is the function to be called in system call mode with args.

err<0 : system call could not be initiated for some reason, try calling syscall(...) direclty.

The return value of the system call is always stored in sctl.rval, see below.

struct SYSTL sctl;

includes following members:

```
sigset_t pending;   stores pending signals, can be used for checking and must be
                    checked for FATAL signals after return from SYSCALLn
long rval;          return value of syscall() or uval in case of a FATAL signal
```

After return from SYSCALLn a test on fatal signals should be performed, something like:

```
if (sctl.pending | _SIG_FTX) ....
```

Inside syscall mode following things are defined:

```
sigset_t *pf=SYSCTL(flags);
```

flags 0 or SCTL_EXP, used to obtain addr of pending flag and to change to or from SCTL_EXP mode if needed

```
SYS_ISPENDING(pf)  returns true if any signal pending
SYS_PENDING(pf)    returns set of pending signals (not its address!)
```

Here is a small example merely to demonstrate the calling sequence. As it is a toy program it doesn't seriously test for fatal signals as it should.

```
/* here is the actual "system call" */
int sys_test(a,b,c)
int a,b,c;
{
  int err,tmout;
  sigset_t *pf;
  printf("entering sys_test(%d,%d,%d)\n",a,b,c);
  if (a<2) printf("sysctl(0) returns %d \n",pf=SYSCTL(0));
  else printf("sysctl(1) returns %d \n",pf=SYSCTL(SCTL_EXP));
  tmout=a<2 ? 100 : -1;
  if (a==4) {testsyscall(); return a;} /* nested syscalls */
  while(0==(SYS_PENDING(pf)&sigmask(SIGQUIT)))
  {
    printf("signals pending: %s\n",psigset(*pf));
    mt_susjb(-1,tmout,NULL);
  }
  if (a==3)
  {
    printf("\nnow test longjmp exit\n");
    raiseu(SIGSEGV,144);
  }
  printf("do normal exit\n");
  return a;
}
/* and here comes the stub functions */
void testsyscall()
{
  int err;
  struct SYSCTL sctl;
  printf("\n&sctl = %d\n",&sctl);
  printf("\ntest plain syscall\n");
  err=SYSCALL3(0,&sctl,sys_test,1,-5,0x100000);
  printf("returns %d, rval %d\n pending :
    %s\n",err,sctl.rval,psigset(sctl.pending));
}
```

As seen in this example, it is possible to do nested 'system calls', but special care is needed to avoid deadlocks; especially this simple example

special care is needed to avoid deadlocks, especially this simple example will hang easily.

Control of signal receiving

For '**%SIG**' handler receiving signals may be disabled by

- setting QDOS_SIGH.sighandler to 0. This will cause the sending job to retry or err.nc
- closing the channel opened for receiving signals.

'**%CSG**' handlers may block signals by the use of sigprocmask(), this will delay the signal after it has been unblocked again or simply ignore them. The sending job can not distinguish whether the signal was processed or blocked or ignored.

For both handlers signal delivery is also controlled by the priority, see below. If the signal is not delivered due to a too small priority, the sending job is returned an ERR_RO.

Priority

is a longword that consists of the actual priorities for each possible state of a job and some flags to control additional features. Thus the definition of receiving priority

struct SIG_PRIOR_R

is slightly different from the sending priority,

struct SIG_PRIOR_S

the common part for both are these bit fields:

```
p.norm ; applied if job is in normal state
p.susp ; ................. suspended
p.wfio ; ................. waiting for io
p.wfjob ; ................. waiting for another job
```

any of them [0-7].

According to the state of the signal receiving job one of these priorites is applied. The job receives the signal only if the priority given by the sending job is GREATER than that of the receiving job. Thus by setting required priority to 0777(octal) a job may inhibit any signals.

Inside an emulated system call p.norm is always applied.

furthermore the sending job may use bits (flags)

p.df_susp;
p.df_wfio;
p.df_wfjob;

if the receiving job is in one of the suspended states and the priority of the sent signal is not big enough to break the suspended state but enough to interrupt the job in the normal state, the according flag is examined and if set, the signal is effectively delivered after the job returns from the suspended state or the I/O call. This does not block the sending job.

In assembler this is a long word with this bit masks:

```
xyzjjjiiisssnnn ;xyz only for use by the sigsending job
```

where

```
n = norm field,
s = suspend,
i = wfio,
j = wfjob,
x = df_wfjob,
y = df_wfio,
z = df_susp
```

The df_* bits should be on when sending signals to C68 programs.

All other bits of priority are reserved and should be cleared, otherwise they could cause ERR_NI or some ugly errors if these bits are assigned in the future.

The following table shows some suggested receiving priority values. For example if your job is doing some work, expecting some signals for communication but can not handle interrupted trap #3 calls a priority like 01611(oct) or 01711(oct) may be appropriate.

0	job isn't doing any useful work, it definitely waits for some signal to proceed
1-3	job is working and expects some signals, use for job communication, asynchronous IO or sockets
4	default value for c68 progs; there is good chance signals will be handled gratefully, ie not terminate the process
5	signals will probably have serious impact on this job
6	signals will probably terminate this job

```
 7  block every signal, report ERR_RO to signaler
```

Signal TRAP #3 calls

Sending signals and setting/examinig timer events is implemented as standard trap#3 io_sstrg, io_edlin calls that are used to send or send and receive message structures.

io_fstrg reads the **struct SIG_INFO** as defined in sys/signal.h into the buffer. This is the easiest way to obtain the address of the **sigvec** routine and the version number of the signal extension.

Messages

the signal extension currently understands three message formats:

```
 struct SIG_MSG, msg.type=M_SIG      to send a simple signal
 struct TMR_MSG, msg.type=M_TIMER    to set/change/cancel a timer event
 struct HINIT_MSG, msg.type=M_HINIT
```

Sending Signals

To send the signal do a TRAP#3 io_sstrg with this parameters:

```
  D0 = #7
  D2.L = #buflen ; sizeof(struct SIG_MSG)
  D3.W = #timeout ; should be >0 if signaling
      ; job==-1 is to work
  A0 = chanid ; signal channel
  A1 = buffer
```

A1 should point at a struct SIG_MSG:

```
 dc.l  '%MSG'    msg.magic
 dc.l  28        msg.len    length of struct
 dc.w  0         msg.type   type and
 dc.w  0         msg.txi    extra info
 dc.l  jobid     msg.jobid
 dc.l  signr     msg.signr
 dc.l  priority  msg.prio
 dc.l  uval      msg.uval   extra parameter that gets passed to the jobs
```

Errors:

```
 err.nc:
 err.bp:  bad message, bad buffer len?
 err.bj:  bad job, or signaling to job #0 or exotic error condition
 err.om:  job supposed to receive signal doesn't have enough stack
 err.nf:  no signal handler established for job
 err.bl:  signal handler found but has bad format or MAGIC id or some
          inconsistent data
 err.ni:  failed due to some unimplemented feature, probably bad setting
          of priority
 err.ro:  priority too small
```

Signal #0 is special, it can be used to test whether a job exists and has established an signal handler.

Getting no error message is no guarantee the signal will get processed. Also the sequence in which incoming signals will be processed is implementation dependent in some cases.

A primitive example how to send signals from basic, setup of timer events works similarly:

```
  100 signr=3: uval=3
  110 jobid=2*65536+2
  120 OPEN#4,'*signal_s'
  130 msg$='%MSG'&lw$(28)&lw$(0)&lw$(jobid)
  140 msg$=msg$&lw$(signr)&lw$(HEX("7fff"))&lw$(uval)
  150 PRINT#4,msg$; : rem : send it !!
  160 DEFine FuNction lw$(x)
  170 LOCal r$,rx
  180 r$=' '
  190 r$(4)=CHR$(FMOD(x,256)) : rx=INT(x/256)
  200 r$(3)=CHR$(FMOD(rx,256)) : rx=INT(rx/256)
  210 r$(2)=CHR$(FMOD(rx,256)) : rx=INT(rx/256)
  220 r$(1)=CHR$(FMOD(rx,256))
  240 RETurn r$
  250 END DEFine
  260 DEFine FuNction FMOD(a,b) : RETurn a-b*INT(a/b)
```

Timer Events

Timer events are identified by their event-id (msg.t_evid) which is unique per job.

EventId #1 is reserved for alarm(), also id's <=10 should be reserved for future use by c68 or other libraries.

setup/cancel timer event : use io_sstrg
parameters as above, message struct see below

setup/cancel and return pending timer event: use io_edlin
a1 should point at the end of the message,
d1=message length,
returns empty buffer or previous event message with t_ticks indicating time units remaining till event that was cancelled/rescheduled

Message structure for timer events is: struct TMR_MSG

```
 dc.l  '%MSG'    msg.magic
 dc.l  40        msg.len
 dc.w  1         msg.type     type is timer
 dc.w  unit      msg.txi      0 = 50/60Hz ticks,
                             1 = seconds
 dc.l  jobid     msg.jobid
 dc.l  signr     msg.signr
 dc.l  priority  msg.prio
 dc.l  uval      msg.uval     extra parameter that gets passed to the job
 dc.l  event_id  msg.t_evid
 dc.l  ticks     msg.t_ticks  #units till interrupt
 dc.l  int       msg.t_int    interval timer if >0
```

Errors:

Same as above, obviously only those that are detected at timer setup time can be reported.

Currently the same primitive time measurement is used for all units. A poll routine is used to count 50/60 Hz ticks, this usually works but gets very inacurate if many disk operations are performed.

Obviously signr can be any legal signr not just SIGALRM.

If msg.signr=0, no event will be generated but any with the given msg.t_evid will be cancelled and (only io.edlin) returned.

Sigvec handler interface

If the handler was setup by the normal procedure, the sigvec routine can be used for handler maintenance and some other things. In c68 programs the address of the sigvec routine can be normaly found in the _sigvec variable.

The parameters are same as in equivalent POSIX calls, currently implemented calls are:

```
  sgv(chid,0): sigcleanup()
  sgv(chid,1,signr,oadr,nadr): sigaction(signr,oadr,nadr)
  sgv(chid,2,how,&omask,&mask): sigprocmask(how,oset,nset), c/e block mask
  sgv(chid,3,&mask): sigpending() examine pending
  sgv(chid,4,signr,uval): raise() signr,uval
  sgv(chid,5,signr,uval) : fraise()
  sgv(chid,6): checksig(), raise pending signals
  sgv(chid,7,signr,&oprio,&prio): c/e priority
  sgv(chid,8,flags,&sctl,&sycfunc,argc,..)
      low level func to implement SYSCALLn macros,
      run syscfunc as emulated system call,
  sgv(chid,9,flags):
      low level func to implement SYSCTL macro
```

Here is the actual implementation of sigaction() as an example:

```
  int sigaction (signo, act, oact)
  int signo;
  struct sigaction *act;
  struct sigaction *oact;
  {
    _oserr=(*(_sigvec))(_sigch,1,signo,act,oact);
    if (_oserr==0) return 0;
    errno=(_oserr==ERR_OR ? ERANGE : EOSERR);
    return (int)SIG_ERR;
  }
```

QED Manual

Version 1.01 - September 1988

Hilversum, Holland.

AUTHOR'S NOTE: The QED program and its documentation are copyrighted. However, you may distribute QED freely on a non-commercial basis provided that you do not alter the QED code or any of the files supplied with it.

1. INTRODUCTION

QED is a QL text editor intended for line-based text files, such as assembler or C source files and the like. Its design is largely based on existing QL text editors, although the main design consideration of QED was to be faster and more compact. For this reason the whole of it was written in assembly language, and this resulted in a program that, despite its power, only occupies about 8K bytes of code. This means that on an unexpanded QL you have more room for your text files. If you have expansion RAM and floppydisks, you will be able to edit files of hundreds of K's with QED while loading and saving still takes only a few seconds to complete.

It must be stressed that QED is only a text editor and **not** a full-blown word processor! Although you can use QED very well to edit "human readable" text, you will not find facilities such as paragraph reformat, justification or footers and the like in QED. So don't throw away QUILL yet...

2. GETTING STARTED

Before using QED, I recommend you to make a backup copy of QED and its complementary files (QED_DOC, QED_HELP and QEDCONFIG_BAS). This can be done with the SuperBASIC (W)COPY command or an automatic copier program.

2.1. Loading

QED can be started in two possible ways:

- in the normal way, using EXEC(_W) <device>QED
- with a command line, using EXEC(_W) <device>QED;'<file name>' (Toolkit II only).

(where <device> stands for the name of the drive containing QED, e.g. mdv1_ or flp1_ depending on your system).

In the first case, QED will ask for the name of the file to be edited, and the size of the workspace to be used. In the latter case, QED will take the filename specified in the command line as the current workfile, and use the default value for the workspace size.

Note that, in both cases, the specified filename must be a full QDOS file name, e.g. FLP2_FRED_ASM.

e.g.:

```
EXEC flp1_QED;'flp2_myfile_c'
```

will load and run QED with **flp2_myfile_c** as the current workfile.

If the specified workfile exists, it will be loaded into QED's workspace and the first lines of it will be displayed on QED's text window. If you have a very long workfile (say 100K or more) there may be a few seconds pause after the drive has stopped while QED is building an internal table.

If you have specified a non-existent file, QED assumes that you want to create a new file so it displays the message "Creating new file" on the bottom of its window. If you have made a mistake when typing in the workfile name, you may use the "R" command to enter a new name.

The default workspace size is calculated as follows: If the specified workfile is an existing one, QED will read the size of the file and allocate a workspace large enough to hold the file plus an additional 4K bytes overhead. This means you can usually add a few hundred lines to the file before the workspace becomes full and you have to save and reload the file.

If the workfile does not exist, QED uses the initial workspace size, which is 12K bytes (but can be re-configured).

If you start QED without a command line, you can override this default and specify the workspace size you like. This can be entered as a decimal number, optionally terminated by a "K". In the first case the number is taken to be the size in bytes, else in Kbytes. Replying with just ENTER will use the default workspace size.

As you already may know, EXEC_W (or EW in Toolkit II) will halt SuperBASIC until you quit QED, and EXEC (or EX in Toolkit II) will leave SuperBASIC running so you can continue with it (possibly starting up a second copy of QED!).

If you have SPEEDSCREEN, please use it because QED will greatly benefit from it! QED and SPEEDSCREEN work the fastest if they are loaded in ROM or expansion RAM. Note that QED is a well-written program that can be put into ROM (using suitable software) or a HOTKEY-file without problems. QED will work well with QRAM, although it will also work well without it!

3. USING QED

3.1. General appearance

The QED window is divided into two areas: the upper region (all except the bottom line of the window) is the "text area" where your editing takes place, and the bottom line of the window is used to show the command line, messages and status information. The status information includes the current line and column position (both starting at 1), the number of lines currently in the textfile and the mode (Insert or Overwrite).

QED always displays a physical line of text on one screen line. If the length of the line is longer than the display width, only part of it is displayed. Moving the cursor outside the window will cause the display to be panned horizontally, so that the other part of a line becomes visible.

The maximum physical length of a line is 254 characters.

3.2. Entering text

QED can operate in two modes: Insert and Overwrite. In Insert mode, any character you type on a line will be inserted at the current position, and any characters further on the line will be moved to the right. Pressing ENTER will split the current line, moving any characters righthand of the cursor to a newly inserted line below the current line.

In Overwrite mode, the character at the cursor position will just be overwritten by the new character you type. Pressing ENTER will only move the cursor to the start of the next line. However, if you type beyond the end of the line the characters you type will just be appended to the line, and pressing ENTER on the last line of the file will generate a new line, just as with Insert mode.

To change the mode, press the F4 key.

QED supports automatic wordwrap at the end of a line. If you reach the right margin of the text (which does not necessarily have to be the same as the rightmost column of the display!), QED performs an automatic newline and places the whole of the partly completed word onto a new line. Also, it is possible to specify a left margin, to which the cursor will move when you reach a new line (either due to an ENTER keypress or a wordwrap).

When you edit a line, QED firstly copies it into an internal buffer. Any changes you make to the line take place in that buffer, not in the text file itself. You can restore the line to its pre-edited state by pressing the ESC key. However, if you move the cursor to another line the newly edited line will be placed back into the text file, thus replacing the original line.

3.3. Cursor control

The cursor can be moved one position by the cursor control keys **LEFT**, **RIGHT**, **UP** or **DOWN**. The display will be scrolled horizontally or vertically if necessary. However it is not possible to move the cursor beyond the start of the line, the 254th column, or the start or end of the text file.

The **SHIFT LEFT** and **SHIFT RIGHT** combination of keys will move the cursor to the start of the previous or next word respectively, while the **ALT LEFT** and **ALT RIGHT** keys will move the cursor to the start or the end of the line respectively.

The **TAB** key generates a number of spaces until the next tab position on the line is reached (which is a multiple of the tab distance). These spaces will be inserted or overwritten depending on the mode.

Note that you can move the cursor beyond the last character of the line, but this will not generate extra spaces at the end of the line. However, if you type a character at this position, spaces will be inserted automatically between the previous end-of-line and the new character.

The **ALT UP** and **ALT DOWN** keys scroll the text one line up or down, while the cursor position on the screen will not move. The **SHIFT UP** and **SHIFT DOWN** keys move the cursor one page up or down, which is useful for quickly moving through the text.

3.4. Deleting text

The **CTRL LEFT** and **CTRL RIGHT** keys delete one character to the left or right of the cursor. The **SHIFT CTRL LEFT** and **SHIFT CTRL RIGHT** keys delete one word to the left or right respectively.

The **CTRL ALT LEFT** key deletes the whole of the current line, while the **CTRL ALT RIGHT** key erases the current line from the cursor to the end.

A CTRL LEFT or SHIFT CTRL LEFT keystroke at the first column of the line will actually delete the "newline" at the end of the previous line: the current line will be appended to the previous line. Note that this will not work on the first line of the text, or if it would generate a line longer than 254 characters.

3.5. Function keys

The function keys F1 to F5 have the following function:

F1: Display HELP information. This is read from a HELP file, so this has to be present on the appropriate medium. For example, if QED uses flp1_QED_HELP as the help file, then the disk containing QED_HELP must be in drive 1. If QED cannot find the HELP file, it will display an error message.

F2: Re-execute last command line.

F3: Allows you to enter and execute a command line.

SHIFT F3: Allows you to edit the last command line.

F4: Change Insert/Overwrite mode.

F5: Redraw the QED display. This is useful if you are multitasking QED without a screen-saving front end program (such as QRAM).

3.6. Commands

Pressing F3 allows you to enter a command line. This consists of one or more commands, terminated by an ENTER keypress. While entering the command line the normal QDOS line editor keys (LEFT, RIGHT, CTRL LEFT and CTRL RIGHT) may be used. The length of a command line may be up to 255 characters, although the display will not allow you to see this number of characters at one time.

You may recall the last command line by pressing SHIFT F3. This allows you to edit the command line again; pressing ENTER will execute it.

Pressing F2 will immediately re-execute the last command line entered; useful if you want to repeat a set of commands after looking at the text.

A command consists of the command name (one or two letters), possibly followed by a numeric or string parameter. A numeric parameter must be a decimal number in the range 1 to 65535 inclusive (there are currently no commands which accept a zero parameter). A string parameter is a sequence of characters starting and ending with a delimiter character. A delimiter character must be non-alphanumeric and may not be a space, semicolon ";" or bracket.

Examples of valid strings are:

```
/fred/
'1/2'
!wombat!
"Hello!"
```

Multiple commands on a single command line must be separated from each other by a semicolon ";". E.g. SA/flp1_myfile_asm/;Q will save the current file to floppydisk 1 under the name "myfile_asm" and then quit QED.

A command may be executed repeatedly by specifying a repeat count before the command name. E.g. the command 4N will move the cursor four lines down in the text file. The repeat count must be in the range 1 to 65535.

A special case of this is the RP specifier: this will repeat the following command indefinitely. E.g. RP E/mdv/flp/ will change all occurrences of "mdv" subsequently found to "flp".

Note that, regardless of the repeat count, a command will always be terminated when an error condition occurs or the ESC key is pressed. This will always return you to the editing mode.

Finally it is possible to combine groups of commands with brackets. A repeat count before the opening bracket will then repeat the group of commands rather than a single command. The commands within a command group may contain their own repeat count, and a command group may also contain further nested command groups. The following (quite silly) example demonstrates this:

RP(4N;12(20P;19N)) will move the cursor position back and forth within the text until it reaches the top or end of the file or you press ESC.

3.7. Description of Commands

3.7.1. Cursor control

There is a set of commands for moving the cursor. Some of these commands are only useful if used in conjunction with other commands and/or repeat sequences, as their effect can also be achieved by using the normal cursor control keys.

The **CL** command moves the cursor one position to the left, while the **CR** command moves it one position to the right.

The **NW** command moves the cursor to the start of the next word, the **PW** command moves it to the start of the previous word.

The **CS** command moves the cursor to the start of the line, the **CE** command moves it to the end of the line.

The **N** command moves the cursor to the next line in the text, the **P** command moves it to the previous line.

The **T** command moves the cursor to the top of the file (first line), the **B** command moves it to the bottom (last line).

The **M** command, which must be followed by a line number (starting at 1 for the first line), moves the cursor to a particular line in the text.

The **RT** command can be used to Return to a particular line. When you issue any command that moves the cursor away from the current line (except N or P), QED stores the current position automatically so that you can return to it later. This is useful if you temporarily want to look at some text

elsewhere in the file (for example to check procedure parameters) but want to return to the current position afterwards.

Notes:

1. QED only remembers the last three positions in the file stored. This means you can, for example, use the M command three times, whereafter three subsequent RT commands will take you back to the former position. But if you do another RT, QED will move to the start of the file as it can no longer remember the "Return line number" which was in use at the time the M commands were issued.
2. In the present version of QED, the stored line numbers will **not** be updated whenever one or more lines are inserted or deleted **before** the stored position. This means that RT will not take you to the correct line in this case. However, if only a few lines are inserted or deleted, the position returned to by RT will not be far away from the original position.

3.7.2. Altering text

The **TY** command enters the characters of its string parameter into the text as if they were TYped from the keyboard. E.g. TY/fred/ will enter the characters "fred" into the text at the cursor position, using insert or overwrite mode as appropriate.

The **DC** command deletes the character under the cursor, in the same way as the CTRL RIGHT key does. The **DW** command deletes the word right from the cursor, as the SHIFT CTRL RIGHT key does.

The **I** command, which may be followed by a string parameter, inserts a new line containing the text of the string into the text between the current line and the previous line. If no string parameter is given, an empty line will be generated.

The **A** command is identical to the I command, except that the new line will be inserted between the current line and the line after it.

The **D** command deletes the current line, moving up all lines below it.

The **J** command joins the current line with the next, appending the text of the next line to that of the current. This will not work on the last line of the text, or if it would generate a line longer than 254 characters.

The **S** command splits the current line at the cursor position, moving the text from the cursor onward onto a new line.

3.7.3. Setting margins and tab distance

The **SL** command sets the left margin equal to the value of the numeric parameter (which must be from 1 upwards). The left margin is the column position to which the cursor will be set after an ENTER keypress or an A, I or S command. In Overwrite mode, the characters left from the left margin will not be erased. It is still possible to move the cursor to the left by using the cursor control keys.

Please avoid setting the left margin to daft values; it should certainly not be set beyond column 254.

The **SR** command sets the right margin equal to the value of the numeric parameter. When you reach the right margin during typing in of the text, QED will automatically perform a newline and move any partially completed word to the next line. Note that this will **not** happen if your typing starts already beyond the right margin, or if the character you type is not the last on the line.

The automatic line wrap can be disabled by setting the right margin to a value greater than 254.

The **ST** command sets the tab distance equal to the value of the numeric parameter. When a TAB keystroke is received, QED will insert or overwrite one or more spaces until it reaches the next multiple of the tab distance.

The distance specified must be in the range 1 to (at most) 253.

The left and right margin and tab distance are initially set to default values. These default values can be configured with the QEDCONFIG_BAS program.

3.7.4. Searching and replacing text

The **F** command searches the text from the cursor position onwards for a target string. If a string parameter is given, this will be taken as the target string, if no parameter is given the target string specified in the last F or BF command will be used. E.g. F/wombat/ will search the text for the string "wombat". Note that the case (upper of lower) of the string is ignored and the surrounding characters are of no importance.

The **BF** command is the same as the F command except that the search will be made from the cursor position backwards into the text.

The **F** and **BF** commands have two immediate versions: the **CTRL DOWN** and **CTRL UP** keys respectively. CTRL DOWN will search forwards for the string given

in the last F or BF command, CTRL UP will search backwards for it.

It should be noted that the F command moves the cursor one position forwards before the search, and BF moves the cursor one position backwards before the search. Hence, if the cursor is already positioned at the start of the target string, the F or BF command will skip to the next or previous occurrence respectively. This is useful if you want to search quickly for a particular occurrence; simply press the F2 or CTRL UP/DOWN key if the occurrence found is not the one you want.

The **E** and **EQ** commands exchange (replace) a particular string by another. The E command exchanges immediately, the EQ command queries first on finding an occurrence (press Y if you really want to exchange).

Two strings must follow, separated by a single delimiter character. E.g. E/cat/mouse/ will replace the next occurrence of "cat" by "mouse". If you want all occurrences to be replaced, use the commands T;RP E/cat/mouse/.

If you do not respond with "Y" on an EQ query, the command will be terminated but the command line will not be aborted unless you pressed ESC. Thus if RP is specified EQ will skip to the next occurrence.

The E and EQ commands follow the same rules for searching as the F command. The first string specified will become the new target string.

3.7.5. Block commands

A block of text consists of one or more complete lines. The start of a block is defined by moving the cursor to the desired line, and then issuing the **BS** command. The end of a block is defined in the same way using the **BE** command. The block is then defined to be the text from the start of the "BS" line up to and including the text on the "BE" line.

If any line is inserted into or deleted from the text (except by a IB command), the block start and end become undefined once more.

Note that the start and end of the block do not have to be specified in strict order, so it is possible to specify first the end and then the start of the block. The BS and BE commands do not check whether the block end is after the block start, the validity of the block is only checked when the next block command is issued.

The **IB** command inserts a copy of the block between the current line and the previous line. It is not possible to insert a block within itself (use IB within copies if necessary).

The **DB** command deletes the block from the text file. The cursor is set at the position where the block has been deleted.

The **WB** command writes the contents of the block to a file, the name of which must be specified in the string parameter.

The **SB** command sets the cursor at the first line of the block and at the top of the display. Note that the RT command may be used to return to the old position.

3.7.6. File commands

In all file commands, any filename specified must be a full QDOS file name (e.g. flp2_fred_asm).

All commands which write text to a file (X, SA and WB) automatically overwrite the file if it already exists, without asking permission.

The **SA** command writes the text to a file. If no parameter is given, SA will use the current workfile name. If a string parameter is specified, this will be taken as the file name, but the current workfile name will not be altered.

E.g. **SA/flp2_myfile_asm/** writes the text to file "myfile_asm" on floppydisk 2.

The **R** command allows you to re-enter QED with a new file name, discarding the old text. The new file name must be specified in the string parameter.

If any changes have been made to the current text file, QED first asks permission to continue. Press Y if you want to re-enter, losing the old text, or any other key to abort the command.

The R command will reclaim the existing workspace and allocate a new one using the default workspace size. If there is insufficient memory for the new workspace, QED will abort with an error message.

The **IF** command inserts the contents of the file specified in the string parameter at the current position. The file will be inserted between the current line and the previous line. If there is not enough room in the workspace, an error message will be generated. In that case, you will have to re-start QED with a larger workspace size.

After the file has been loaded, there may be a few seconds delay during which QED is rebuilding an internal table.

3.7.7. Miscellaneous commands

The **Q** command quits QED without saving the text. If any changes have been made to the text, QED will ask confirmation. Press Y if you want to leave QED, losing the changes.

The **X** command writes the text back to the file (using the current workfile

name) and then quits QED. It is in fact the same as the commands SA;Q. If any file error occurs (not found, drive full etc.), QED remains running.

The **U** command cancels any changes made on the current line. It does in fact the same as the ESC keypress. It has been included for compatibility with other QL text editors.

The **SH** command displays the current status of QED. This includes the name of the workfile, the current string used for the F and BF commands, the tab distance, left and right margin, block start and end line, and the workspace size and usage.

3.8. Error messages

If during editing or command execution an error occurs, QED will display an appropriate error message. In most cases this is self-explaining, but two cases might need further explanation:

"No room for text"

There is not enough room in the editing workspace left for what you are trying to do. You must save the text first (using SA) and then reload it using the R command (which allocates a new workspace large enough for the file plus 4K overhead).

"No room for line table"

QED maintains an internal table to keep track of the length of each line. When this table becomes full, QED rebuilts it using a larger table size. This is normally transparent to the user, but if there is insufficient memory in the QL to do it then QED will display this message and inhibit any further editing. You should save the text using the SA or X command, then do something to get more free memory, and then reload the file.

4. CONFIGURING QED

QED allows you to re-configure many of its startup parameters, such as window size and position, display colours, default margins etc. This can be done using the QEDCONFIG_BAS configuration program.

The configuration program is started by entering the SuperBASIC command:

```
LRUN <device>QEDCONFIG_BAS
```

(where <device> is mdv1_, flp1_ etc. as appropriate).

Once the program has been loaded, it will ask for the name of the medium containing the copy of QED to be configured. E.g. if you have a copy of QED in microdrive 1, you must enter mdv1_, and so on.

The configuration program will then read and display the currently installed values. You can then select the parameter to be altered with the UP and DOWN keys, and modify its value with the LEFT and RIGHT keys. The LEFT key will decrease the value by one, the RIGHT key will increase it. An exception to this is the name of the HELP file: pressing LEFT or RIGHT allows you to enter the new HELP file name, which must be terminated by an ENTER keypress.

The following parameters can be configured:

- The left and right margin and the tab distance to be used on startup, or after a R command.
- The initial workspace size in Kilobytes. This is the workspace size used whenever the default is specified on startup or the R command is used, and the workfile does not exist.
- The colour and width of the border of the QED display.
- The PAPER and INK colour of the initial display (the display used on startup).
- The PAPER and INK colour of the text window.
- The PAPER and INK colour of the status and report line.
- The PAPER and INK colour of the command line.
- The horizontal character size. This corresponds to the first parameter of the SuperBASIC CSIZE command: 0 for 6 pixels, 1 for 8 pixels, 2 for 12 pixels and 3 for 16 pixels. Note that the command and status line must at least contain 55 characters, so the horizontal character size of this line will never be greater than 8 pixels, even if you specify 12 or 16 for the text window. Also, QED always uses MODE 4 for its display, switching the MODE on startup if necessary.
- The name of the HELP file. This is the file displayed when F1 is pressed. Note that this is just a normal text file that can be edited with QED, so you can easily create your own HELP file and install it using this configuration program. Is is best however to avoid lines longer than 55 characters, as QED might not be able to display them fully (the HELP display has no end-of-line wrap).

For the PAPER and INK parameters, consult the QL User Guide if necessary. For the PAPER (background) colour a range of 0 to 255 is valid, for the INK

(foreground) colour only 0 to 7 as "stippled INK" text is somewhat difficult to read! The configuration program relies on your own confidence on the choice of colours, so don't use silly combinations like green INK on white PAPER and the like. Remember that QED always uses 4-colour MODE!

When you have finished, press ENTER to continue with the window size and position configuration. A window is then displayed with colours, border, size and position equal to those of the window to be configured.

The window can be re-positioned with the LEFT, RIGHT, UP and DOWN keys. One keypress will move it two pixels in horizontal direction or one in vertical direction. The window can be re-sized with the ALT LEFT, ALT RIGHT, ALT UP and ALT DOWN keys. One keypress reduces or enlarges the window by one character column or line as appropriate. The minimum size is 55 columns and 5 lines.

Any change you make will be displayed; also the current window width and height in **characters** (not pixels) will be displayed, with the pixel coordinates of the top left-hand side below it.

When you are satisfied with the window size and position, press ENTER. The program then finally asks whether you want to install the new parameters or not. Press Y to install them or N if you want to quit without installing.

APPENDIX 1: COMMAND SUMMARY

A1.1. Immediate commands

TAB	Tabulate (i.e. insert/overwrite a number of spaces)
ENTER	Split and generate new line (insert mode) Move cursor to new line (overwrite mode)
LEFT	Move cursor left one character
RIGHT	Move cursor right one character
SHIFT LEFT	Move cursor left one word
SHIFT RIGHT	Move cursor right one word
CTRL LEFT	Delete left one character
CTRL RIGHT	Delete right one character
SHIFT CTRL LEFT	Delete left one word
SHIFT CTRL RIGHT	Delete right one word
ALT LEFT	Move cursor to start of line
ALT RIGHT	Move cursor to end of line
CTRL ALT LEFT	Delete line
CTRL ALT RIGHT	Erase to end of line
UP	Move one line up
DOWN	Move one line down
ALT UP	Scroll one line up
ALT DOWN	Scroll one line down
SHIFT UP	Move up one page
SHIFT DOWN	Move down one page
CTRL UP	Search backwards (see "BF" command)
CTRL DOWN	Search forwards (see "F" command)
ESC	Restore line to its pre-edited state
F1	Display HELP information
F2	Re-execute last command line
F3	Enter and execute command line
SHIFT F3	Re-edit and enter last command line
F4	Toggle Insert/Overwrite mode
F5	Redraw display

A1.2. Extended commands

In the following:

n indicates a number in the range 1 to 65535;

/s/ indicates a string starting and ending with a delimiter character;

/s/t/ indicates two strings separated from each other by a single delimiter character (which must be the same as the character which introduces **s** and terminates **t**).

The characters **[ ]** indicate an optional parameter.

A[/s/]	Insert line containing **s** after current line
B	Move to bottom of file
BE	Mark block end
BF[/s/]	Backwards find
BS	Mark block start
CE	Move cursor to end of line
CL	Move cursor one position left
CR	Move cursor one position right
CS	Move cursor to start of line

D	Delete current line
DB	Delete block
DC	Delete character under cursor
DW	Delete word right from cursor
E/s/t/	Exchange **s** into **t**
EQ/s/t/	Exchange but query first
F[/s/]	Forwards find
I[/s/]	Insert line containing **s** before current
IB	Insert copy of block
IF/s/	Insert file **s**
J	Join current line with next
M **n**	Move to line **n**
N	Move to next line
NW	Move to next word
P	Move to previous line
PW	Move to previous word
Q	Quit without saving text
R/s/	Re-enter editor with file **s**
RP	Repeat until error
RT	Return to previous line
S	Split line at cursor
SA[/s/]	Save text to file
SB	Show block on display
SH	Show status
SL **n**	Set left margin
SR **n**	Set right margin
ST **n**	Set tab distance
T	Move to top of file
TY/s/	Type string **s**
U	Undo changes on current line
WB/s/	Write block to file **s**
X	Exit, writing text back

APPENDIX 2: FILE FORMAT AND MEMORY USAGE

QED is designed to handle line-based QDOS textfiles, which are files containing lines of printable characters terminated by a LF character.

QED restricts the maximum length of a line to 254 characters; if you load a file containing longer lines these will be truncated to 254 characters. QED will give a warning message if it has done so.

The maximum size of the file you can edit depends on the amount of RAM in your QL (QED loads the whole file into RAM), but you must also ensure that your file does not contain more than 32768 lines (which is more than enough for most purposes). QED will however read files containing more than 32768 lines, but will ignore the extra lines. Again a warning message is displayed if this has happened.

Apart from the QED job code and data space (which are 8K and 1.5K respectively in the current version), QED uses two areas of RAM, both of which are allocated in the QDOS Common Heap area.

The first area is the textfile's workspace. This holds the text file in its original form, without any additional data. This has the advantage that LOADing and SAVing of the text file can be done very quickly using QDOS string I/O calls. However the disadvantage is that changing the length of a line near the beginning of a long textfile involves moving a large block of memory, which may slow down editing. (Note that editing the line itself does not take place in the text file but in a buffer, so this will not be slowed).

The QED code is optimised to speed this up as much as possible, but on an internal RAM machine it may take a second to enter a line near the start of a 160K textfile. If you find this too slow, remember that it is better to code a large program in parts which can be linked together by a linker (if possible), rather than in one very large source file. This will also save compiling/assembling time.

When a text file is loaded, QED will also allocate a line length table. This holds the length (one byte) of each line, enabling a particular line to be found quickly. In fact, if you want the cursor to be positioned at a particular line, QED will first determine the shortest path to take (which can be down from the top of the file, up or down from the current position, or up from the bottom of the file) and then search backwards or forwards for the line you want.

When the file is loaded, QED will allocate a line length table large enough for the text file itself plus an additional 256 lines. If during expansion of the file the table becomes full, it is reclaimed and a larger table

of the file the table becomes full, it is reclaimed and a larger table built using the contents of the text file (this may take a few seconds). If the attempt to re-allocate the table fails due to shortage of memory, an error message "No room for line table" is displayed and any further editing will be prevented. However it is still possible to SAVE the file as the line length information is not required for SAVing.

APPENDIX 3: REVISION HISTORY

This appendix describes the revisions carried out on QED since its first release.

V1.00 (August 1988)

First release version.

V1.01 (September 1988)

SHIFT F3 command added to re-edit last command line.

PW, NW and DW commands added (move to previous word, move to next word and delete word right from cursor respectively).

Bug in DB command fixed (caused display to be scrambled on certain occasions).

Bug which caused garbage to be displayed at end of text when too-long lines were read fixed.

F5 key now also redraws border of display.

END OF MANUAL

SROFF File Format

INTRODUCTION

This document will not be of interest or relevance to the average C programmer. It is included as part of the C68 package as the information is of interest to System Programmers, and has not been made widely available within the QDOS or SS programming communities.

It is also intended to provide extensions to the originally defined standard to support interactive debuggers. This will be accompanied by upgrading the LD linker to recognise these extensions. The LD linker will also be upgraded so that i can be used with other programming languages, and not just C68 as at present.

RELOCATABLE BINARY FORMAT

The Sinclair Relocatable File Format (or SROFF) was defined as the format of files that are suitable for linking to produce binary code. In the C68 system the _o files produced by AS68 and all the libraries are in this format.

A relocatable object file consists of a sequence of modules, each of which is a sequence of bytes terminated by an **END** directive (see below). It should have a QDOS file type of 2 though this will not be enforced by the linker. Interspersed with the sequence of bytes can be directives from the list below. A directive is a sequence of bytes beginning with the hex value FB.

When otherwise unmodified by a directive, a byte indicates that it should be inserted at the current address and the address should be stepped by 1. The special directive **FB FB** inserts the value **FB** in this way.

Note that bytes are overwritten on (not added into) the byte stream, so that if several sections are located at the same address, it is possible to overlap (or even interleave) their contents. This is useful for Fortran block data.

In the following syntax definition, **<word>** s and **<longword>** s need not be word aligned: they just follow on from the preceding data with no padding bytes.

A **<string>** consists of a length byte (value range 0-255), followed by the bytes in the string. A **<symbol>** is a **<string>** of up to 32 chars. A symbol should start with a letter (A-Z), a dot or an underline (N.B. the original SROFF definition did not allow an underscore at the start of a symbol but this has now become commonly accepted) and the other characters may be letters, digits, dollar, underline or dot.

DEFINITION OF A SECTION

A **SECTION** is a contiguous block of code output by the linker. Each section has a name, and any source file can add to one or more of the sections. A module's contribution to a section is called a subsection.

The linker will arrange that each section or subsection will start on an even address, by inserting one padding byte if necessary. The value of this byte will be undefined.

Note that if a module returns to a section, this is part of the same subsection and the linker will not re-align on a word address.

When a section name is used in an **XREF** command the address of the start of the subsection is used.

the subsection is used.

Note that section names are maintained separately from symbol names (and module names), so there can be a section, a symbol and a module all with the same name without any danger of confusion.

DIRECTIVES

The following lists the possible directives in ascending value. See later for rules governing permissible orders of directives.

SOURCE syntax: FB 01 <string>

The **<string>** in this directive indicates information about the source code file from which the following bytes were generated. This directive should only appear at the start of a module (ie at the start of a file or immediately after an **END** directive).

The string will start with the module name which may be followed by a space followed by a field of further information about such things as the version number or the date of creation or compilation. The string should contain only printable characters and be no longer than 80 characters.

This module name should conform to the syntax of a **<symbol>** defined above, and may be used by the linker to identify individual modules within a library (see later). The module name can be generated from a QDOS or SMS filename, but it is recommended that the device name is first stripped off.

COMMENT syntax: FB 02 <string>

The **<string>** in this directive is a line of comment. It will have no effect on the binary file, but should be included at some suitable point in a link map. The string should contain only printable characters and be no longer than 80 characters.

ORG syntax: FB 03 <longword>

This indicates that the bytes following the directive are to start at the absolute address given in the parameter. This applies until the next **ORG** , **SECTION** or **COMMON** directive.

SECTION syntax: FB 04 <id>

This indicates that the bytes following the directive are to be placed in the relocatable section whose name was defined in a **DEFINE** command with the **<id>** value specified.

This applies until the next **ORG** , **SECTION** or **COMMON** directive.

OFFSET syntax: FB 05 <longword>

This directive updates the output address: the longword specifies the address relative to the start of the current subsection or the latest ORG directive.

The parameter is unsigned, so the offset may not be negative.

XDEF syntax: FB 06 <symbol> <longword> <id>

This indicates that the symbol whose name is the **<symbol>** is defined to be the value given in **<longword>** , relative to the start of the subsection referred to by the **<id>** . Note that an **<id>** of zero defines the symbol to be absolute.

See the description of **DEFINE** for the definition of **<id>** .

XREF syntax: FB 07 <longword><truncation-rule> { <op><id> } FB

This indicates that the result of an expression involving user symbols or other relocatable elements is to be written into the byte stream. Note that this command does not overwrite existing bytes, but appends new bytes to the output.

The **<longword>** parameter defines an absolute term for inclusion in the expression to be evaluated by the linker.

The **<truncation-rule>** parameter is a byte which defines the size of the final result and the circumstances in which the linker might give a truncation error, or the mode in which truncation should occur (undefined bits must be set to zero). These are the effects of setting each bit:

a) If bit 0 is set, the result is one byte.
If bit 1 is set, the result is a word.
If bit 2 is set, the result is a longword
Only one of these three bits may be set.

b) If bit 3 is set, then the number is signed.
If bit 4 is set, the number is unsigned.
Only one of these two bits may be set.
See notes below

c) If bit 5 is set, the reference is PC relative, and the relocated current address (ie the address to be updated by this directive) is to be subtracted before the truncation process.

d) If bit 6 is set, runtime relocation is requested (for longwords only). The address of the longword is included in a table generated by the linker which can be used by a runtime loader. See later for the

the linker which can be used by a runtime loader. See later for the format(s) of this table.

After the **<truncation-rule>** is a sequence of terms for the expression. **<op>** is a one-byte operator code and can be 2B for "+" or 2D for "-". **<id>** is a symbol or section name id as defined in the **DEFINE** directive. The special **<id>** code of zero refers to the current location counter (ie the address updated by this directive).

The final **FB** byte terminates the sequence of terms in the expression.

As an example of the use of the signed/unsigned bits, consider a value which must be written out as a word value; the signed/unsigned bits are interpreted as follows:

	resulting value
< FFFF8000	always out of range
FFFF8000 to FFFFFFFF	illegal if 'unsigned' bit is set
00000000 to 00007FFF	always allowed
00008000 to 0000FFFF	illegal if 'signed' bit set
> 0000FFFF	out of range

DEFINE syntax: FB 10 <id> <symbol>
FB 10 <id> <section name>

This directive is used in conjunction with **XDEF** , **XREF** , **SECTION** and **COMMON** directives. The directive defines that the **<symbol>** or **<section name>** may be referenced by the 2-byte **<id>** . A **<section name>** has the same syntax as a **<symbol>** .

Note that positive nonzero **<id>** values refer to symbols and negative **<id>** values refer to section names. This directive must appear before the **<id>** value is used in any other directive.

If within a signle SROFF module two **<id>** values are used to refer to the same symbol, or if one **<id>** value is reassigned to another then the effects are undefined.

COMMON syntax: FB 12 <id>

This directive is identical to the **SECTION** directive except that it informs the linker that the section is to be a common section so that references to this section id in different object modules refer to the same memory location.

Within the same object module multiple additions to the same section will be appended together as for an ordinary section.

When different modules create common sections of differing size, the linker should create a section equal in size to the largest one.

END syntax: FB 13

This directive marks the end of the current object module. If the file contains only one module, then this will appear at the end of file.

DIRECTIVE ORDERING

Mandatory Rules

Within a relocatable object file the following rules should be applied to the ordering of directives within an object module.

1. A **SECTION** directive (or **ORG** or **COMMON**) must appear before any data bytes in the module.
2. A symbol or section's **<id>** must be defined in a **DEFINE** directive before it is used in any other directive.

The ordering of other directives is at the discretion of authors of compilers or relocatable assemblers, though it will normally be dictated by the source code.

BNF DEFINITION OF A SROFF FILE

This BNF uses **{ }** to mean 0 or more repetitions of an item.

<relocatable object file> = <module> { <module> }

<module> = SOURCE { <chunk> } END

<chunk> = <header> <body>

<header = { <header command> } <section command>

<header command> = COMMAND | XDEF | DEFINE

<section command> = SECTION | ORG | COMMON

<body> = { <data byte> | <body command> }

<body command> = OFFSET | XDEF | XREF | DEFINE | COMMENT

LIBRARY FORMAT

The traditional format for a library has simply been a relocatable object file as described above, that contains more than one module. Such a library can be created by appending smaller libraries or object files. The SLB librarian provided with C68 provides an easy tool for manipulating modules

in such libraries.

When a linker searches a library it checks each module to see if it resolves any external references. If so that module will be included in the link.

The SLB librarian (v2.10 onwards) and LD linker (v2.00 onwards) also support an enhanced library format that allows much faster linking. This enhanced format basically consists of a header section that details what symbols are externally visible within each module, and the location of that module in the library. The remainder of the library is then in the traditional format. The format of this header table is:

8 bytes	Preset to "<<XDEF>>"

This is then followed by repeating entries for each symbol that is globally visible. These entries will be in the order that they occur within the SROFF part of the library file. The format of these entries is:

long	File offset within the library to the start of the module that contains this symbol. A value of zero is used to indicate the end of the XDEF area.
string	The symbol name (as a zero terminated C style string). An additional zero byte will be added if necessary to ensure that this symbol ends on an even boundary. The symbol is in principle case significant although both SLB and LD have runtime parameter options to ignore the case of external symbols.

This format is very similar to that of the XDEF area used within RLL libraries and binary program files. The difference is that the offset is to the start of the module containing the symbol.

It is also possible for a symbol to be defined more than once in different modules. The linker always searches this table forwards from the point at which the last module was included.

RELOCATION TABLE

The relocation table is generated by the linker as described earlier. The exact format depends on the linker used. QDOS does not include any standard facility to handle such relocation tables so the relevant code needs to be included in the user program.

In the case of C programs this is handled by the start-up module that is always automatically linked in right at the beginning. It is important that the start-up module corresponds to the relocation table format. In the case of C68 this is the crt_o library module if you use the **LD** linker supplied with C68, and the qlstart_o library module if you use the GST linker.

The GST Linker (LINK)

The table consists of a series of longwords giving the address relative to the start of the program. The table is terminated by a longword containing a negative value.

C68 programs that are linked with the GST linker should always include the QLSTART_O module as the first one in their program. This module will assume that the relocation table is in the GST linker format.

The C68 Linker (LD)

The C68 linker LD v1.xx introduced a new format that is the same as the one used on the Atari ST (for the format used by LD v2.00 onwards refer to the RLL_DOC file). The new format is slightly more complicated, but it results in the final table being much smaller in size (typically 25-30% of the size produced by the GST linker). This can make the size of the fil holding the final object program to be 10-15% smaller than the same file linked with the GST linker.

The table starts with a longword giving the address relative to the start of the program of the first address to be relocated.

The remainder of the table consists of one byte entries giving the displacement of the next address to be relocated relative to the previous one. The special value of 1 is used to mean advance the location pointer by 254 without actually doing a relocation. The table is terminated by a byte of value 0.

As an optimisation, the LD linker assumes that after program initialisation, the space used by the relocation table will be reclaimed for use as "Unitialised Variables" space. This reduces the runtime memory requirements of C68 programs.

C68 programs that are linked with the LD linker should always include the CRT_O module (or the CRESPR_O one if they are writing resident procedures) as the first one in their program. This module will assume that the relocation table is in the LD linker format.

In practise, the LD linker wll add the CRT_O module by default, so it is normally unecessary for the programmer to take any special action to get the CRT_O module incuded in their program.

CHANGE HISTORY

This is a short summary of the changes that have been made to this document. The intention is to make it easy for users who are upgrading to find any new information

find any new information.

25 Apr 94 DJW Minor changes and corrections for the 4.13 release.

30 Dec 94 DJW Updated the description of library files to include the new format supported by SLB v3.00 and LD v2.00.

Technical Reference

INTRODUCTION

This document is intended for those who wish to write programs and/or libraries that need to work with the C68 compilation system. It therefore documents some of the system interfaces that are internal to the C68 system.

It also covers any topic details for which are not thought to be appropriate to any other document.

The topics covered include:

- Data Formats
- Assembler Language Interface
- Namespace Pollution and Name Hiding
- Program Start-up parameters
- Memory Allocation
- Unix I/O emulation
- Run Time libraries
- Hardware Floating Point Support

1. DATA FORMATS

This section covers the detail of the formats of the different data types within C68.

It is important to note that in the case of all multi-byte data types, the address in memory will always be aligned on an even memory address. For fields within complex data items this means that (invisible) padding fields may be added to achieve this.

int

In the QDOS C68 implementation, the **int** keyword has the characteristics of a **long** as described below.

char

This is a 8 bit value held in a single byte. If not specified, then **char** is treated as signed. It can hold values in the range:

-128 to 127 if signed

0 to 255 if unsigned.

short

This is a 16 bit value held in two bytes. It can hold values in the range:

-32768 to 32767 if signed

0 to 65535 if unsigned

long

This is a 32 bit value held in 4 bytes. The data type **int** is also of this size and characteristics. It can hold values in the range:

-2147483648 to 2147483647 if signed

0 to 4294967295 if unsigned

pointers

Pointers of all types are held as 32 bit values held in 4 bytes. They can therefore be stored in types of **int** or **long** without loss of accuracy.

float

The internal representation of **float** in C68 uses the IEEE 32 bit format (but see below, however, for the format prior to C68 Release 3). The IEEE 32 bit representation of a floating point number is equivalent to the C structure:

```
struct IEEE_FLOAT {
  int sign-bit : 1;
  int exponent : 8;
  int mantissa : 23;
};
```

The exponent is biased by a value of 127. The most significant bit of the mantissa is implicit (i.e. not actually present) and is always set.

It can hold values to 6 or 7 significant digits in the range:

+/- 10E-37 to +/- 10E38

C68 Releases 1 and 2

The implementation of C68 before version 3 held floats as 32 bit values in Motorola Fast Floating Point format. This represents a **float** as if it were the following C structure.

```
struct MFFP_FLOAT {
  int mantissa : 24;
  int sign-bit : 1;
  int exponent : 7;
};
```

The exponent is biased by a value of 63.

The mantissa is organised as a number that when multiplied by 2 to the power of the exponent becomes the required value. The value of the exponent is chosen so that the most significant bit of the mantissa is always set.

double

The internal representation of **double** in C68 uses the IEEE 64 bit format (but see below, however, for the format prior to C68 Release 3). The IEEE 64 bit representation of a floating point number is equivalent to the C structure:

```
struct IEEE_DOUBLE {
  int sign-bit : 1;
  int exponent : 11;
  int mantissa : 52;
};
```

The exponent is biased by a value of 1023. The most significant bit of the mantissa is implicit (i.e. not actually present) and is always set.

It can hold values to 15 or 16 significant digits the range:

+/- 10E-307 to +/- 10E308

C68 Release 1 and 2

In releases of C68 before Release 3, the **double** keyword was equivalent to the **float** one, and numbers were held in the same 32 bit Motorola Fast Floating Point Format as mentioned above under **float**.

long double

This is a new floating point data type defined in ANSI C to allow for even greater precision that given by the **double** data type. The current inmplementation of C68 recognises this data type, but treats it with the same accuracy as the **double** data type.

It is intended that a future implementation will support this data type with more accuracy by using the IEEE 80 bit representation of floating point numbers. The IEEE 64 bit representation of a floating point number is equivalent to the C structure:

```
struct IEEE_LONG_DOUBLE {
  int sign-bit : 1;
  int exponent : 15;
  int mantissa : 64;
};
```

The exponent is biased by a value of 16383.

2. ASSEMBLER LANGUAGE INTERFACE

Programmers may write assembly language modules for inclusion in C programs provided that these modules adhere to the object code linkage and function calling conventions described below:

The C68 compilation system contains its own assembler **AS68** . It is not necessary, however, to use this assembler if you you would prefer to use an alternative one. The output from the AS68 assembler conforms to standard Sinclair SROFF format, so any assembler that produces this format can be used. The problem, however, with many existing assemblers is that they commonly suffer from one or both of the following:

1. They do not treat labels as case dependant.
2. They truncate external references to less than the 31 characters supported by C68/AS68.
3. External symbols are all converted to upper case.

This does make them unuseable with C68, but one needs to be aware of they way the assembler handles each of the above cases.

External names

It is necessary that the name of the subroutine be made visible outside the assembler module. Also, any variables to be used by the C program must similarily be made visible. The C68 compiler adds an underscore character to any externally visible C name to ensure that there is no possible clash with reserved words in the AS68 assembler. This character must be added

explicitly at the assembler level for names to be correctly visible at the C level.

NOTE See also the section on "Name Hiding" later in this document.

The current QL linkers, including the one supplied with C68, ignore the case of external references. It is intended that in a future release of C68, the linker will be case sensitive, so it is highly advisable to keep the case of externally visible names consistent.

It is worth noting that ANSI has reserved any name starting with the underscore character for use by the implementors. Application programmers use them at their peril unless explicitly instructed to do so by the library writers! As the compiler always adds one underscore anyway, an underscore at the C level translates to two underscores at the assembler level. If you have any names that are not meant to be visible to the application programmer, then either do not start them with an underscore character, or use at least two underscore characters.

Parameter Passing

Parameters are passed following the standard C convention of pushing them onto the stack in right to left order. The calling module is also responsible for removing these parameters from the stack when the call returns.

C68 also follows the standard C definition whereby certain parameter types are 'widened' when they are passed on the stack. They thus take up more room than one might guess at first sight. The exact widening depends on whether there is an ANSI prototype in scope or not (if there is no ANSI prototype in scope then K&R rules are used). The parameter types that are affected by this widening action are as follows:

Parameter Type	**Passed as ANSI Prototype**	**Passed as K&R Protoype**
char	short	int
short	short	int
float	float	double

Register Usage

The C user does not need to be aware of register usage. However, the assembler programmer has to know what registers are used by C68 generated code. The normal usage of registers by C68 generated code is as follows:

D0-D2	Scratch registers for temporary results
A0-A1	
D3-D7	Used for holding register variables
A2-A5	
A6	Used as stack frame pointer.
A7	Stack pointer

The user can also specify a lower number address register to be used as the frame register by using the **-frame=** n runtime option with C68. If this is done, then any address registers between the one specified as the stack frame pointer and the stack pointer (A7) are not used by C68 generated code.

Return Values

Results are returned from functions in registers. They are returned as follows:

D0.B	A character value (8 bits)
D0.W	A short value (16 bits)
D0.L	A long value (32 bits) or a pointer or a float (32 bit)
D0.L and D1.L	A double (64 bits)
D0.L, D1.L and D2.L	A long double (96 bits). Note however, that as delivered, C68 currently does not utilise this option as it treats **long double** as being equivalent to **double** .

NOTES.

1. In C68 an int can be either 16 bit or 32 bit. The default in the QDOS implementation is 32 bit, but this can be overridden by a runtime option to C68. This should be done with care as the issued libraries assume 32 bit int.

Section Names

The C68 Compilation System allocates section names as follows:

TEXT	code
DATA	initialised data area

UDATA uninitialised data areas

It is not mandatory that you follow these conventions in your own assembler routines, but it is recommended unless you have a good reason to do otherwise.

In addition the following two additional areas can be set up automatically by the linker:

BSS Relocation information.

RLSI Run-Time library symbol information.

The linker always places the UDATA section at the end of those specified by the user, and then the BSS and RLSI sections. It then assumes that after any relocation is done, the information in the BSS and RLSI sections will no longer be required, and the space can be added to that in the UDATA section. The size of UDATA is calculated on this basis.

Relocation Information

This is a table generated automatically by the linker to hold relocation information. Its format is:

long — First address needing relocation relative to start of the program. Relocations are always applied to long words. If zero, then no relocation is required.

byte — Next address that needs relocating relative to previous address. A value of 1 means add 254 to the previous address, but do not do any relocation at this new address.

A byte of 0 terminates the table.

This is different to the format that was traditionally generated by the GST Linker.

Run-Time Library Symbol Information

This is a table that is only generated if the Linker has been told that Run-Time libraries are to be used. Its format is described later in the section on Run-Time Libraries.

Differences from Lattice C

The assembler level interface used by C68 is very close to that used by QL lattice C. In most cases libraries can be written to be compatible with both compilers. There are however a few differences:

- C68 demands more registers be preserved than Lattice C. Routines that preserve enough registers for C68 will always have saved enough for Lattice C.
- The method of passing structures is different. Lattice C passes a pointer to a structure, while C68 passes a copy of the structure.
- Lattice C always follows the K&R rules for widening parameters. It is important therefore that you do not use any of the data types 'char', 'short' or 'float' as parameter types if you are trying to write libraries that are portable between Lattice C and C68.

3. NAMESPACE POLLUTION AND NAME HIDING

The ANSI C standard states that the C namespace should not be polluted by names that are defined in header files that the programmer has NOT included. This is necessary so that the C application programmer does not have to worry about whether any externally visible names in his program conflict (probably without his knowledge) with any names in the library unless he has specifically included a header file that adds these names to the programs namespace.

To illustrate what this means, take the case where a C application programmer decides to include a routine called **read()** within his program. There is also a routine called **read()** defined within the **unistd.h** header file tht emulates the Unix **read()** system call. This latter version of **read()** is called internally within the library from many other library routines. What ANSI C states is that the version of **read()** in the library and the version of **read()** in the user program are to be treated as different functions if the programmer has not explicitly included the **unistd.h** header file.

To implement this capability, it is necessary therefore for library routines that are called internally within the library have a "hidden" name which is different to the name seen by the C application programmer. This means that it does not matter if the programmer inadvertantly includes in his program a routine that has the same name as the public name of a library routine.

The way that this is implemented in C68 is that any routine which is called internally within the libraries has a **#define** statement in the appropriate header file to add an underscore to the public name. This new name with the underscore added is now a private name that is only visible if the appropriate header file is included. The C programmers call to the public

name is therefore changed without his knowledge to a call to the hidden name. All library routines are written so that they always include all appropriate header files so that calls between libray routines always go via this hidden name. If you are calling such routines from assembler it is necessary to add this extra underscore explicitly.

While this is fine in practise, it is quite common in C programs that are ported from other machines that the programmer has not included all the header files he should have in his program, and has instead relied on C's "implicit declaration" facility. This "implicit declaration" occurs when the C compiler encounters a function call, and there is not already a definition of declaration of that function in scope, then the function has an implicit return type of 'int', and all parameters are passed using K&R parameter promotion rules. In this case as the appropriate header file has not been included, the compiler does not know that it should convert the public name to a hidden name.

As this practise is so common, this is got around by including dummy routines in the library that simply map the public name to the private name. An example might be

```
    .text
    .globl strcpy
strcpy:
    jmp _strcpy
```

This simply means that the routine with the private name '_strcpy' will still be found if it is called by the public name 'strcpy' even if the relevant header (string.h in this case) is omitted. If the header IS used, then the reulting code would make a direct call via the private name which is more effecient, both in terms of speed and code size.

4. PROGRAM START-UP PARAMETERS

When a C68 program starts up then before control is passed to the users **main()** process, a number of standard actions are taken:

Redirection of stdin, stdout and stderr

The command line is examined to see if any of the standard files have been re-directed. If so this is acted on.

Channels passed as parameters

The stack is examined to see if any channels were passed as parameters. If they were, then they are allocated to stdin, then to channels 3 onwards, and finally to stdout.

Normally, if channels are passed on the stack, then a close() call in C only closes the file at the C level, and does not really close the underlying channel. This is to stop child jobs messing up the screen channels of the parent job. If the top bit is set for any channel passed on the stack, however, the QDOS close will be performed. This facility is currently intended for the "child" end of pipes.

Default Directories

A number of routines in the C68 libraries will allow directory names to be defaulted. If a number of c68 compiled jobs are chained together then the default directories of each slave job are inherited from the master. The master job will obtain its values from the settings in SuperBasic.

This inheritance factor is important as it means that if a library call is used to change one of the default directory settings, then this is remembered and passed on **without** changing the setting that is current at the SuperBasic level.

For releases of C68 prior to Release 2.01, the program determined on start-up if it had inherited default program directories by examining the stack. If it had inherited directories, then it will find the following sequence (following any program parameters):

```
 2 byte flag  $4A $FB
 C string     Default Data Directory
 C string     Default Program Directory
 C string     Default Destination (Spool) directory
 1 byte       $00
```

From Release 2.01 onwards, the mechanism changed to share that used for Environment variables. The Default Directories were instead stored as the Environment variables PROG_USE, DATA_USE and SPL_USE. These are passed to daughter jobs just like any other Environment Variables.

If on start-up it is found that default directories have not been inherited from the parent job, then the values are obtained from SuperBasic.

Environment Variables

A number of routines in the C68 libraries will allow environment variables to be examined and/or altered. If a number of c68 compiled jobs are chained together then the environment variables of each slave job are inherited from the master. The master job will obtain its values from the settings in SuperBasic.

This inheritance factor is important as it means that if a library call is

used to change one of the environment variables, then this is remembered and passed on **without** changing the setting that is current at the SuperBasic level.

A C68 program will determine on start-up if it has inherited environment variables by examining the program stack. If it has inherited environment variables, then it will find the following sequence (following any program parameters):

```
 2 byte flag  $4A $FC

             Pointer to Environment Variables area. This consists of a
 long        sequence of C strings of the form "NAME=value". They are
             terminated by a NULL byte.
```

If on program start-up it is found that no Environment Variables have been inherited from the parent job, then the values are obtained from SuperBasic.

5. MEMORY ALLOCATION

The C68 system maintains a private heap for each C68 program. QDOS system calls are used both to maintain the private heap, and to obtain/release memory from QDOS. The space for this heap is allocated dynamically from the QDOS Common heap as described below.

Initial Allocation

This is allocated by the **crt_o** start-up module. An area is allocated that is large enough to contain both the initial memory requested, and also the program stack.

The amount of memory allocated at this stage is controlled by the combination of the **_mneed** and **_stack** global variables.

The value for the stack required is first obtained from the **_stack** global variable. The command line is then examined to see if the runtime **=** option is used in the program parameters then. This value, if present, overrides the value stated in the **_stack** global variable (only increases allowed). A minimum of 1Kb is allocated even if the programmer and user both specified less than this.

The value for the initial allocation of data space is obtained from the **_mneed** global variable. The command line is then examined to see if the runtime **%** option is used in the program parameters. This value if present over-rides the value stated in the **_mneed** global variable. It also has the side effect of setting the **_memmax** global variables to the same value. This means that the initial program allocation is equal to the value specified, and the program cannot allocate any additional memory.

Placement of the Program stack

The program stack is placed at the top end of the initial memory allocation. This means that the stack grows down towards the users data area. If the stack overflows then the user program's data areas are corrupted before any other areas on the Common Heap. This should mean that in the event of stack overflow the User Program will often fail before corrupting the system. Because of the dire consequences of stack overflow, some library routines (**stackcheck()** and **stackreport()**) are provided to allow user Programs to anticipate and check for possible problems in this area.

Additional Allocations

Additional memory allocations are made when there is no free memory in the current private heap. To avoid excessive fragmentation of the QDOS Common heap space is added to the private heap of the C68 program in chunks of at least the size specified in the **_memincr** global variable.

If the C68 program finds that a call to release memory frees all of one of the areas allocated from the QDOS Common Heap, then this area is returned to QDOS.

6. UNIX I/O EMULATION

One of the features of the C68 emulation is its extensive emulation of Unix system calls. The Level 1 I/O under C68 corresponds to the normal Unix I/O interface. C68 provides library routines which mimic the Unix system call interface.

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. The file descriptors are mimicked under QDOS by having an array of UFB structures (these are defined in the libc_h file supplied with the source to the library). These structures contain the underlying QDOS information (such as the QDOS channel) corresponding to any given file descriptor. They also contain flags that describe the mode in which this file should be handled.

The UFB structures are pointed to by the global variable

```
struct UFB *_ufbs;
```

It should not be necessary for the average user to ever access these structures directly. However if you do, you need to be aware that certain system calls such as **open() , dup** , and **dup2()** can cause them to be moved

in memory. It is recommended therefore that you access the information they contain using supplied libray calls such as **getchid()** and **fcntl()** .

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. **7. RUN TIME LIBRARIES**

This facility is still under development, and details are subject to change. This section will be completed when the RLL facility is ready for use.

8. HARDWARE FLOATING POINT SUPPORT

8.1 Overview

The original QDOS and SMS systems did not have hardware floating point units (FPU) and so the QDOS and SMS systems have no built in support for hardware FPU. There are, however, an increasing number of systems that run compatible operating systems that **do** have hardware FPU.

It has therefore been decided to define how hardware FPU would be supported on such systems and provide an implementation of this definition. The design is such that the definition should be generic enough in nature so that anyone who wants to implement hardware FPU support will be able to do so. The implementation is not limited to C68 in any way.

Thanks must go to **George Gwilt** who has been the one who has developed and implemented the core code. This code is distributed as part of the C68 release with his permission. It is also available separately independently of C68. Other key particpants were **Dave Walker** working on the C68 related aspects and **Simon Goodwin** working at providing SuperBasic extensions that exploited the FPU.

8.2 Save/Restore FPU context

Possibly the most important omission in the QDOS and SMS operating systems as as FPU support is concerned is that they have no built in facilities for saving and restoring the FPU context on task switches. This therefore has to be done using additional software. The code to implement this has been been developed by George Gwilt and is included as part of the standard C68 release (from release 4.22 onwards).

8.3 Floating Point Support Package

The other part of the George Gwilt implementation is a generalised version of the Motorola FPSP (Floating Point Support Package). There is a core FP instruction set that is implemented across all the Motorola processors. There is then additional FP instructions that are implemented in some Motorola processors but not others. The FPSP packages are software implementations for each processor of the missing instructions.

As each processor has different missing instructions, Motorola provide different FLP Support Packages. What George Gwilt has done is provide a common interface to all of them. Where a particular processor implements the instruction in hardware then that is called directly and where it implements it in software then that is called. The interface code automatically detects which Motorola processor you have and automatically sets up the correct FP Support package. This means that on QDOS the programmer can call the FP Support routines always without the need to do anything different according to processor type.

8.4 C68 FPU Support

The C68 system supports the use of hardware floating point. There are two implementations available:

- The first is implemented completely within the C68 libraries in such a manner that the user does not have to take any special considerations to get hardware floating point support. If you use floating point within your program, then the C68 system code will automatically look at runtime to see if hardware floating point is present (i.e. the SP_FPSAVE_BIN file has been loaded), and if so it will use it. If there is no hardware support for floating point then the previous software implementation will be used instead.
  The implementation described above is not as fast as would be the case if hardware floating point instructions were generated inline by the compiler (due to the overheads of making library calls). However it does have the advantage of generating portable object programs while still giving a useful performance boost to floating point operations when the hardware is present.
- The second implementation allows for inline generation of floating point instructions. This implementation provides maximum speed at the

expense of producing programs that are not capable of being run on systems that do not have hardware floating point support.

Both of these implementations are supported by the C68 system.

The support was actually implemented incrementally across a few different C68 releases as follows:

C68 Release 4.20	C68 libraries enhanced to look for FPU hardware and to attempt to use it if present. This was an interim implementation that did not depend on proper FPU support being added to QDOS.
C68 Release 4.21	C68 compiler enhanced to provide an option to generate FPU instructions in-line, and the GWASS assembler added to the release to assemble such code (the default ACK asembler cannot handle FPU instructions). This was limited by the fact that there was no support in QDOS for saving/restoring the FPU context on task switching or correctly handling FPU exceptions.
C68 Release 4.22	The George Gwilt support code for correctly adding FPU support to QDOS added to the C68 release. The C68 FP support routines in the LIBC_A library upgraded to exploit this.

8.5 System Variables for FPU support

The following new System Variables are associated with the support of hardware floating point. They have been formally registered with Tony Tebby (as the author of QDOS and SMSQ) so that they do not get used for any other purpose.

Name	Offset	Size	Description
sys_fpu	$d0	byte	Set to indicate the presence of hardware floating point support. Values used are: -ve No hardware Floating Point (or use of hardware floating point is suppressed). 0 Unknown if FPU hardware present. This value would be the default value on current systems. 1 68881 or equivalent 2 68882 or equivalent 4 68040 6 68060 In addition, if the Floating Point Support Package is loaded (Library version) then bit 3 is also set (i.e. 8 is added to the above values).
sys_mmu	$d1	byte	Set to indicate the type of MMU present in the system. Values used are: 1 68851 3 68030 4 68040 or 68LC040 6 68060 or 68LC060
sys_fpzs	$d2	word	Maximum length of FSAVE area.
sys_fpsl	$d4	long	Address of save area list.
sys_clfp	$d8	long	Address to access the FPSP (**F** loating **P** oint **S** upport **P** ackage). Set to 0 if not present.
sys_fpxx	$dc	long	Currently unused.

If you are using George Gwilt's code for adding FPU support to QDOS then the sys_fpsl is set up when you first use the FPU or any of the FPSP routines. The other variables are set up when the SP_FPSAVE_BIN file is loaded.

8.6 Thor FPU support

The Thor 20 machine apparently included FPU support and used some system variables in the $d0-$df range, but in a slightly different way. However the Thor always seems to set the variable at $d4, so if this is already set then the FP support package supplied with QDOS will refuse to load, and C68 will therefore revert to software FP implementation.

If anyone has more information on how the Thor supported FPU then I would welcome it so that we can:

1. Document the usage for future reference.
2. Ensure that we can correctly distinguish between the Thor FPU system and the new one defined above.

8.7 Further FPU Information

For more detailed definitions of the low level interfaces to the FPU support please refer to George Gwilt's documentation. A copy of this will be included on the C68 Documentation disks. Please note that in the case of

any discrepancy between what is defined here and George's documentation it is likely that George's is correct.

8.8 Assemblers that support FPU instructions

If you want to program the FPU in assembler you will need as assembler that supports the additional op-codes required for FPU support. George Gwilt has developed the GWASS assembler that has this capability and allowed it to be freely distributed. A copy of the GWASS assembler is included with the C68 release. It is run by the C68 CC front-end automatically instead of the AS68 assembler whenever you ask C68 to generate inline FPU code. You can off course also use it independently of the C68 system.

AMENDMENT HISTORY

The following is a checklist of any important changes that have taken place to this document. It is intended to help users who are upgrading from one release of C68 to another to rapidly identify changed information.

21 Oct 93	Changed the description of parameter widening to conform to the new ANSI compatible mode introduced with C68 Release 4.04.
10 Nov 93	Added fact that external C symbols have underscore prepended by compiler.
25 Apr 94	Minor cosmetic changes for the 4.13 release of the C68 system.
14 Aug 94	Added a new section on Namespace hiding within the various C68 libraries.
20 Sep 95	Added the beginning of a section on hardware floating point support.
03 Jan 97	Updated definition of FPU support to bring it inline with the George Gwilt implementation now that it all appears to be working.

TOS Emulator for QDOS

A short note on the TOS emulator for QDOS. 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 currently one of the best 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 should contact Dave Walker (NOT Jeremy Allison as he is busy on other work). 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 new version of Lattice C (version 5) that has recently been released for the Atari ST.

Issue Notes for Release 4.22

INTRODUCTION

This document contains the Release Notes for this release of C68. It gives you some guidance as to the parts making up the system and any last minute thoughts.

If you must get started as quickly as possible (and do not intend to use the tutorial) then it it is still recommended that you read the STARTING_DOC file as this gives you basic guidance on running the **C68** environment.

Also, before you dive in it would be particularly useful to read the C68 overview (in OVERVIEW_DOC) and the description of the C68 programming environment (in QdosC68_DOC). The rest of the documentation can then be read as required. Eventually you are likely to want to print yourself copies of most of the documentation for reference purposes.

NEW FEATURES OF THIS RELEASE

The key new feature of this release relative to the 4.20 release are:

- Some new optimisations in the code generated that will result in smaller and faster code.
- Bug fixes to the library.

There are also some options in this compiler that are brand new, but should still be treated as experimental in nature. They relate to the support for inline generation of floating point instructions, and the associated GWass assembler that can assemble such code.

NEW FEATURES OF THE 4.20 RELEASE

This release is intended primarily as a major maintenance release of the C68 system. Its prime purpose is to clear all bugs reported to date. However, due to the long time since the last release there is also quite a bit of new functionality. The key major new features that are in this release are:

- Support for Unix style Signal handling. This is comprehensively described in the file SIGNALS_DOC.
- Support for hardware floating point when it is present in the machine running C68 programs that use floating point arithmetic (**and system variable offset $d0 set non-zero**).
- A new alternative maths library (the Cephes library) that appears to be faster and more accurate than the existing library. At this release the old version of the maths library is retained in case of any problems being encountered with the new one.
- Restructuring of much of the library documentation. It is hoped that this will make it easier to use. In particular a comprehensive cross-reference is now supplied that covers all the main C68 libraries.

There are also a host of other small improvements. While none of them in themselves are particularily significant, it is hoped that the sum of them will produce significant benefits to C68 users.

If you want more details of the changes, as well as information regarding bugs found and cleared then please refer to the document CHANGES_DOC. This document gives a complete history of what has changed between the various releases of C68.

FEATURES NOT IN THIS RELEASE

Features NOT in this release

It is also worth noting that there are many C68 related developments that are well under way that did NOT make it into this release, and will thus be features of future C68 releases. Examples of such developments are:

- Run Time Link libraries. This has been outstanding for some time now. It was hoped that this development would be completed for this release, but it was not possible. The documentation for the RLL system is provided for those who are interested in how it might work (albeit it might still change slightly before final release).
- Improved Environment Variable support. This development will provide new capabilities for Environment Variable support. In particular it will make this support available to languages other than C and SuperBasic.

The intent is that many of these above capabilties may be released at a future date. Initially this will be via Electronic Mail and Bulleting Board systems.

ISSUE DISKS

The C68 system is issued on a number of 720Kb disks. There is the RUNTIME set that contains all you need to use C68, the DOCUMENT set that covers all aspects of using C68, and the SOURCE set that contains the C source of the components of the C68 system. The contents of these disks are as follows:

RUNTIME_1	The main C68 system disk containing the various passes of the compiler, the header files and the libraries. There are also a number of useful utility programs.
RUNTIME_2	Utility files needed at boot stage. In a future release this will become the RLL version of the RUNTIME_1 disk.
RUNTIME_3	Additional libraries and header files for use with C68. Also includes all files required if BOOT'ing from C68 issue disks.
DOCUMENT_1	Documentation covering general aspects of the C68 system, and the main programs making up the compilation suite. This is all held in QUILL files as all QDOS users will have a copy of Quill.
DOCUMENT_2	Documentation for the libraries and extra utilities. This is held in QUILL format.
SOURCE_1	The source for the main programs making up the C68 system. These are C68MENU, CC, CPP, C68, AS68, LD, MAKE and SLB. These are all provided in compressed ZIP format.
SOURCE_2	The source for the main libraries supplied with the C68 system. These are all supplied in compressed ZIP format. There is also (space permitting) the source of a small number of the utility programs.
SOURCE_3	The source for additional libraries supplied with the C68 system. These are all supplied in compressed ZIP format. There is also (space permitting) the source of a small number of the utility programs.

There are also a number of additional disks (with more being worked on) that are related to the C68 system. These will not be appropriate for all users, so they are not provided as part of the basic set. These disks all contain the object code, the documentation and the source code for their particular topic area. The current release number of each disk is given so that existing users of these disks can tell which disks have been recently upgraded.

C_TUTOR "C Tutorial v2.0"

An excellent C tutorial for those just starting with C. This is annotated with notes that relate it more directly to the QL/C68 environment. Please note, however, that it covers the more traditional K&R C rather than all the features that are part of ANSI C. Despite that it is well worth working through for new users.

GNU RCS "GNU Revision Control System 5.6"

This is used to help keep track of changes between different releases of source in a controlled manner. It is highly recommended to any serious programmer whatever the programming language they are using.

Note that if you only have 720Kb floppy disk drives you may find this package a bit cumbersome to run - it really requires a hard disk or high capacity floppy drives to get the most out of it. Note also that it requires you to have the GNU DIFF package mentioned below.

GNU DIFF "GNU Diff v2.4"

This is a series of programs that handle detecting the differences between files. This package is needed if you want

to use the GNU RCS pacakage.

GNU UTILS "GNU Text Utilities v1.9"

This is a series of about 20 Unix style utility programs associated with manipulating text files (e.g. cat, expand, pr, sort, wc etc). One disk contains the programs, and the other one the documentation and source code.

PROGTOOLS1 "C Programming Tools 1" (v1.2)

Some useful C programming tools:

CPROTO: Automatically build header files containing function prototypes for your programs.

INDENT: Reformat C programs to your own specific standard layout.

UNPROTO: Convert programs written in ANSI style C to ones in traditional K&R C.

QPTR "QPTR Companion Disk v3.05"

Material to help with using LIBQPTR library. Includes a tutorial from Tony Tebby and some useful utilities from various sources.

The documentation supplied on this disk assumes that you already have the QPTR documentation. Without this documentation you will not find this disk to be of much use.

LIBCPORT "C68 CPORT Support Library v1.36"

This is a special library of utility programs and Support Routines for use with the CPORT SuperBasic to C converter. It requires the use of C68 v4.10 or later.

The LIBCPORT_A library provides the support routines that are called by CPORT converted programs. The CFIX utility is a post-processor to CPORT that automates much of the additional work that was necessary to get the output of CPORT to compile satisfactorily with C68. Also included are guidelines on using CPORT with C68.

This disk is aimed primarily at those who have a copy of the CPORT software (which is sold by Digital Precision). However SuperBasic programmers converting to C may find it useful as it contains a library of routines that emulate many of the SuperBasic keywords.

LIBCURSES "C68 Curses Library v1.24"

This is a complete implementation under QDOS of a Unix SVR4 curses library. It will be of particular interest to those trying to port programs from Unix that use curses.

Please note that the documentation supplied does not document all the standard CURSES routine - merely the QDOS specific parts of the implementation.

ELVIS "Elvis for C68 v1.8"

This is a port of the Elvis editor and its associated utilities for use with QDOS. Elvis is a clone of the Unix "Vi" editor.

DOCUMENTATION

There is extensive documentation for all parts of the C68 system. The DOCUMENT_1 disk contains the general documentation and that relating to the main programs in the C68 suite. The DOCUMENT_2 disk covers the libraries and all the utility programs. You will still find, however, that you also need a C reference book that covers the standard C libraries.

README_DOC This document. **You should always read this first whenever you obtain a new release of the C68 system.**

CHANGES_DOC A log of the changes that have occurred between the different releases. **Also covers known bugs and new facilities under development.**

OVERVIEW_DOC An overview of the C68 system.

STARTING_DOC A "Quick Start" guide to using C68.

QDOSC68_DOC Details of how QDOS and the C68 programming environment work together.

TECHREF_DOC A technical reference on some of the C68 interfaces. Particularily relevant if trying to write assembler routines to interface to C68.

SIGNALS_DOC A description of the Signal support sub-system for QDOS.

SROFF_DOC A description of the SROFF file format.

The following describe the main programs supplied as part of the C68 system:

MENU_DOC Manual for the C68_Menu graphical front-end to using the C68 system.

CC_DOC Manual for CC command that drives the main programs in the C68 Compilation System.

CPP_DOC	Manual for the CPP pre-processor.
C68_DOC	Manual for the C68 compiler.
AS68_DOC	Manual for the AS68 assembler.
LD_DOC	Manual for the LD linker.

The following give details of routines in the libraries supplied with the C68 system:

LIBINDEX_DOC	The main index and cross-reference for the rest of the C68 library documentation.
LIBANSI_DOC	Short Reference for ANSI defined C library routines.
LIBUNIX_DOC	Short Reference for POSIX and UNIX compatible routines.
LIBC68_DOC	Short Reference for C68 specific routines and LATTICE compatible routines.
LIBQDOS_DOC	Short Reference for QDOS specific routines.
LIBSMS_DOC	Short Reference for SMS specific routines
LIBM_DOC	Short Reference for Maths library.
LIBQPTR_DOC	Short Reference for the Pointer Environment Library routines.

There will then also be a series of files containing the documentation for each of the utility programs supplied as part of the C68 system:

CMDINDEX_DOC	An index to the utility programs available for use with C68. This includes both those provided with the basic C68 set of disks plus those available on additional disks. For each command, the disk on which it can be found will be indicated.
xxx_DOC	manual for the xxx command. Refer to the CMDINDEX_DOC file to get a list of the utilities provided.

If you produce additional documentation, find errors in the current documentation, or improve the current documentation, then please submit your efforts to the C68 issue co-ordinator for inclusion in future distributions of the C68 Compilation System.

ZIP FORMAT FILES

You may well find a number of files on the issue disks whose filenames end in **_zip** . This will apply in particular to disks that contain source code. If present, these are archives that contain a number of files stored together and compressed using the ZIP utility. If you want to use these files they need to be unpacked first by using the UNZIP program. UNZIP is supplied on the RUNTIME 3 disk.

For those not familiar with the UNZIP program there is also a little SuperBasic program UNZIP_BAS on the same disk to simplify the process of unpacking archived files.

EDITORS

The C68 standard distribution includes the QED Public Domain editor.

Editors are very much a personal preference, so if you already use an editor that is capable of editing C source files, then feel free to continue using it. Suitable editors are THE EDITOR from Digital Precision; SPY from ARK; QD3 from Jochen Merz; or the editors for other programming languages.

TUTORIAL

If you are new to C then it is well worth getting the additional C_TUTOR disk. This contains an excellent tutorial for learning C together with plenty of sample programs to illustrate the various points. As you work through the tutorial you will compile and run these examples, so you will also get practical experience in using the C68 environment.

ISSUE CO-ORDINATION

The issue of the QDOS C68 Compilation System is (currently) being co-ordinated by:

Dave Walker
22 Kimptons Mead
Potters Bar
Herts
EN6 3HZ
United Kingdom

Tel: +44 1707 652791 (answering machine during working hours)
Fax: +44 1707 850937 (may not always be switched on)
Mobile: +44 973 382248

Email: d.j.walker@x400.icl.co.uk
or itimpi@msn.com
Web Site: http://www.chez.com/davewalker/

Please report any problems encountered in using the C68 system to the above

Please report any problems encountered in using the C68 system to the above address. If the problem is merely a "how to use/do" type question then it can probably be answered immediately. If it is a genuine fault in the software, then it will be added to the list of "Known Problems", and hopefully it will be cleared in the next C68 release. In the case of genuine bugs, it is useful to have a simple method of reproducing the fault.

KNOWN BUGS & RESTRICTIONS

Refer to the CHANGES_DOC file for details of any known bugs at this release. This is kept up-to-date with any reported problems. It will also list any restrictions or previously reported bugs that have been cleared in the current release.

If you encounter any problems that you think may be due to bugs in the C68 software then please do not be afraid to report them. Bugs will only get fixed if they are reported.

Bug reports should be sent to the issue co-ordinator with as much evidence as you can supply. In particular it is useful to know how to reproduce the bug on a regular basis so that it can be investigated. In addition, please always quote which release of C68 you are using.

DISTRIBUTION CHANNELS

The main distribution channel for the latest C68 releases is now via the World Wide Web. The site for obtinaing both news on C68 releases and copies of the software is:

http://www.chez.com/davewalker/

This site will always contain the last complete release of C68, plus any patches and fixes that have been developed since that release. It will also contain other items of software that are not part of the C68 release, but are relevant to C68 users.

In addition to the main distribution channel via the web mentioned above, **C68** is also often distributed via various other means such as the QL BBS network. Many QL related Public Domain libraries also hold copies of the C68 system. **However please be warned that unless the library owner is careful to keep up to date with the material on the web site the versions available from the PD libraries may not always be the latest one available.** The web site mentioned earlier is always the reference point for checking what is the latest release of C68.

FURTHER DISTRIBUTION OF C68

There is no restriction to further copying of the C68 system for QDOS as long as it is not done for commercial purposes (barring a reasonable charge to cover copying costs). In particular Bulleting boards and Public Domain libraries are encouraged to distribute C68.

It is convenient if you always make certain that copies of C68 are clearly labelled as to which release they refer. It is also convenient if all the files that make up a particular disk are distributed together. This makes it easier to handle any enquiries regarding problems and/or updates.

With the standard distribution on 720Kb disks, if you directory each disk you will see a version number displayed which is the number by which this particular release is known.

ACKNOWLEDGMENTS

The components upon which the QDOS C68 Compilation System is based has come from a number of sources - in particular users of the MINIX operating system. Thanks must go to all those who were prepared to put their efforts into the Public Domain.

CP: Copy Files

NAME

cp - copy file(s)

SYNOPSIS

cp [-i] [-k] [-m] [-v] sourcefile targetfile

cp [-i] [-k] [-m] [-r] [-v] [-z] [-s sorttext] sourcefilelist directory

DESCRIPTION

cp copies a file to another file, or copies a series of files to another directory.

If only two parameters are present after the flags and the second filename is not that of a directory, then a simple file copy is done.

When the last name is a directory, then all preceeding names are assumed to be files to be copied to that directory. If more than two parameters are found then the last parameter must be the name of a directory.

By default, **cp** will only copy files, and will ignore hard directories. If the **-r** flag has been used, then directories and all their contained files will also be copied. The **cp** program can therefore be used with the **-r** flag to copy complete directory trees.

The **cp** copying process will attempt to ensure that the copy of the file has

The **cp** copying process will attempt to ensure that the copy of the file has the same time and date as the original. On version 2 filing systems, **cp** will also attempt to preserve the version number information.

The flag options that apply to **cp** are:

-i If the targetfile already exists, then query the user before overwriting it.

-k If a copy fails while in progress, keep the uncompleted copy of the failed file.

-m Move files rather than simply copying them. The effect is that as long as the copy succeeds without error, then the original file is deleted.

-r Recursively copy any (sub)directories found.

-v Verbose mode. Report on progress of the copying process, and give statistics at the end.

-z Only usable if **-r** also specified. This option tells **cp** not to copy any completely empty directories.

-s This option only applies when copying multiple files. If present, then files will be sorted according to the '**sorttext** ' field supplied. If not present then files are copied in the order they are found in the directory of the source device. The values allowed for '**sorttext** ' follow the standards used by QRAM and QPAC, and are any combination of:

sorttext
- N or n Sort on ASCII name
- U or u Sort on file useage
- S or s Sort on file size
- D or d Sort on file date
- T or t Sort on file time

Upper case=ascending, lower case=descending.

COPYRIGHT

AMENDMENT HISTORY

30 Aug 93 DJW Added the **-r** options.

14 Oct 93 DJW Added the **-z**, **-s** and **-m** options.

FGREP: Search Files

NAME

fgrep - search files for a character string

SYNOPSIS

fgrep [options] string [file..]

DESCRIPTION

fgrep (fast grep) searches a file (or list of files) for a character string. By default it prints the lines containing that string.

If the character string contains spaces (or any characters with special meaning to C programs) then it should be enclosed in quotes.

If no filename is supplied, then standard input is assumed.

If list of files is specified, then they will each be searched in turn. **fgrep** also recognises wildcard symbols in filenames. **?** represents a single undefined character and ***** represent a sequence of undefined characters (could be none). **fgrep** will expand any names containing wildcards to give the approriate list of filenames. If a list of files is being searched, then the filename will be printed before each line listed.

The options that apply to **fgrep** are:

-c Print only a count of lines containing the string.

-i Ignore upper/lower case distinction during comparisons.

-l Print the names only of files with matching lines. The name will only be repeated once regardless of the number of times the string matches lines in the file.

-n Precede each line by its line number in the file (line numbers start at 1).

-v Print all lines EXCEPT those that contain the string.

-x Print only lines which match the string in their entirety.

-y The same as -i option.

-e<string> Search for a special string. This version allows strings to start with the - symbol.

-f<filename> Take the list of strings from the specifed file.

GREP: Search for Patterns

NAME

grep - search file[s] for pattern[s]

SYNOPSIS

grep [-options] ... [expression] [filelist] ...

DESCRIPTION

This program will find a string specified by a regular expression in a file or group of files. The following options are recognized:

-v All lines but those matching are printed.

-c Only a count of the matching lines is printed.

-l The names of the files with matching lines are listed (once) separated by newlines.

-n Each line is preceded by its line number in the file.

-h Do not print filename headers with output lines.

-y All characters in the file are mapped to upper case before matching.

-e <expr> Same as a simple expression argument, but useful when the expression begins with a "-".

-f <file> The regular expression is taken from the file. If several regular expressions are listed (separated by newlines or |s) then a match will be flagged if any of the regular expressions are satisfied.

-e and **-f** are mutually exclusive. If **-f** is given, any regular expression on the command line is taken to be a filename.

Regular expressions are composed of the following:

A **^** matches the beginning of a line.

A **$** matches the end of a line.

A **** followed by a single character matches that character. In this way a " ***** " will match an asterisk, a " **\.** " matches a period, etc. The following sequences are special:

\b backspace (^H)

\n linefeed (^J this is not the same as $)

\r carriage return (^M)

\s space

\t tab

**** backslash

A . matches any character.

A single character not otherwise endowed with special meaning matches that character.

A string enclosed in brackets **[]** specifies a "character class". Any single character in the string will be matched. For example " **[abc]** " will match an a, b or c. Ranges of ASCII character codes may be abbreviated as in " **[a-z0-9]** ". If the first symbol following the **[** is a **^** then a "negative character class" is specified. In this case, the string matches all characters except those enclosed in the brackets (i.e., **[^a-z]** matches everything except lower case letters). Note that a negative character class must match something, even though that something cannot be any of the characters listed. For example: " **^$** " is not the same as " **^[^z]$** ". The first example will match an empty line (beginning of line followed by end of line); the second example matches a beginning of line followed by any character except a **z** followed by end of line. In the second example a character must be present on the line, but that character cannot be a **z** . Note that ***** , **.** , **^** and **$** are not special characters when inside a character class.

A regular expression followed by a ***** matches zero or more matches of the regular expression.

Two regular expressions concatenated match a match of the first followed by a match of the second.

Two regular expressions separated by a **|** or a newline match either a match for the first or a match for the second.

The order of precedence is **[]** then ***** then concatenation then **|** then newline.

EXAMPLE

The command line:

grep -n ^[a-z][a-z]*[\s\t]*.*([^;]*)[^;]*$ <files>

creates a cross reference of a large C program. "<files>" should be replaced with a list of the modules to be searched. Grep's output will show all subroutine declarations in all the listed files. In addition, every output line will be preceded by both the name of the file in which the line was found (this is automatic if more than one file is searched) and by the appropriate line number (the -n causes line numbers to be shown).

The regular expression is interpreted as follows: beginning of line (^) followed by one or more occurrences on any character in the range a to z ([a-z][a-z]*), followed by either a space or a tab repeated zero or more times ([\s\t]*), followed by any character repeated zero or more times (.*), followed by a open parenthesis ((), followed by any character except a semicolon repeated zero or more times ([^;]*), followed by a close parenthesis ()), followed by any character except a semicolon repeated zero or more times ([^;]*), followed by end of line ($).

BUGS

All features of the unix version of grep are supported except the -s, -x and -b options and the metacharacters **(,) , +** and **? .**

Options, if present, must be grouped together in the second position on the command line. The first character of the group must be a -. Unless the -f option is given, the next argument is always taken to be the expression. If -f is present then the third argument is the name of the file containing the expressions.

Beware of spaces or tabs in the expression, even if your compiler supports quoted arguments. CP/M will object to ^I anywhere on the command line. Use \s for spaces and \t for tabs to be safe.

Some of the command line switches do mutually exclusive things (like -ef and -eh). If you try to trick grep into doing something it is not supposed to, the output will be undefined.

Grep's execution speed varies as a function of the type of expression being parsed. The speed will vary as follows (listed fastest to slowest):

- Simple expressions anchored to beginning of line (^<expression>).
- Expressions matching literal strings.
- Expressions including character classes ([]).
- Expressions including closure (*).

AUTHOR

Allen Holub. Code originally published in Dr. Dobb's Journal, October 1984.

QDOS port by Erling Jacobsen.

QDOS SPECIFIC COMMENTS

The expression is best put in quotes, to prevent the run-time initialisation code from interpreting it as a wildcard, and expanding it.

You could also remove the wildcard-expansion facility from grep_c (by removing any reference to cmdexpand()), and recompile. Of course, you then lose the possibility of specifying a filelist using wildcards.

PR: Prepare for Printing

NAME

pr - convert text files for printing

SYNOPSIS

```
pr [+PAGE] [-COLUMN] [-abcdfFmrtv]
  [-e[in-tab-char[in-tab-width]]] [-h header]
  [-i[out-tab-char[out-tab-width]]] [-l page-length]
  [-n[number-separator[digits]]] [-o left-margin]
  [-s[column-separator]] [-w page-width] [file...]
```

DESCRIPTION

This manual page documents the GNU version of pr. Pr prints on the standard output a paginated and optionally multicolumn copy of the text files given on the command line, or of the standard input if no files are given or when the file name '-' is encountered. Form feeds in the input cause page breaks in the output.

OPTIONS

+PAGE Begin printing with page PAGE.

-COLUMN Produce COLUMN-column output and print columns down. The column width is automatically decreased as COLUMN increases; unless you use the -w option to increase the page width as well, this option might cause some columns to be truncated.

-a Print columns across rather than down.

-b Balance columns on the last page.

-

-c Print control characters using hat notation (e.g., '^G'); print other unprintable characters in octal backslash notation.

-d Double space the output.

-e -e[in-tab-char[in-tab-width]] Expand tabs to spaces on input. Optional argument in-tab-char is the input tab character, default tab. Optional argument in-tab-width is the input tab character's width, default 8.

-F , **-f** Use a formfeed instead of newlines to separate output pages.

-h header Replace the filename in the header with the string header.

-i -i[out-tab-char[out-tab-width]] Replace spaces with tabs on output. Optional argument out-tab-char is the output tab character, default tab. Optional argument out-tab-width is the output tab character's width, default 8.

-l -l page-length Set the page length to page-length lines. The default is 66. If page-length is less than 10, the headers and footers are omitted, as if the -t option had been given.

-m Print all files in parallel, one in each column.

-n -n[number-separator[digits]] Precede each column with a line number; with parallel files, precede each line with a line number. Optional argument number-separator is the character to print after each number, default tab. Optional argument digits is the number of digits per line number, default 5.

-o -o left-margin Offset each line with a margin left-margin spaces wide. The total page width is this offset plus the width set with the -w option.

-r Do not print a warning message when an argument file cannot be opened. Failure to open a file still makes the exit status nonzero, however.

-s -s[column-separator] Separate columns by the single character column-separator, default tab, instead of spaces.

-t Do not print the 5-line header and the 5-line trailer that are normally on each page, and do not fill out the bottoms of pages (with blank lines or formfeeds).

-v Print unprintable characters in octal backslash notation.

-w page-width Set the page width to page-width columns. The default is 72.

RM: Remove Files

NAME

rm - remove files or directories

SYNOPSIS

rm [-[f][i][r]] filelist

DESCRIPTION

rm removes (deletes) files or directories. **filelist** is a list of the files or directories to be removed.

The options that apply to **rm** are:

-f Force remove without prompting the user. No errors will be reported if a file could not be found, or if it could not be removed for some reason.

-i Interactively remove files. Query the removal of each file.

-r Recursively delete directories. For this option **filename** should be the name of a directory. If the device does not support directories, then specifying a device name will cause all files on that device to be deleted. For safety's sake, it is advisable to use this option in conjunction with the **-i** option.

TO DO

To allow wildcard to be used within the filenames given to the **rm** command.

COPYRIGHT

SED: Stream Editor

NAME

sed - the stream editor

SYNOPSIS

sed [-n] [-g] [-e script] [-f sfile] [file] ...

DESCRIPTION

SED copies the named files (standard input default) to the standard output, edited according to a script of commands.

An **-e** option supplies a single edit command from the next argument; if there are several of these they are executed in the order in which they appear. If there is just one **-e** option and no **-f** 's, the **-e** flag may be omitted.

An **-f** option causes commands to be taken from the file "sfile"; if there are several of these they are executed in the order in which they appear; **-e** and **-f** commands may be mixed.

The **-g** option causes sed to act as though every substitute command in the script has a g suffix.

The **-n** option suppresses the default output.

A script consists of commands, one per line, of the following form:

[address **[** , address **]]** function **[** arguments **]**

Normally sed cyclically copies a line of input into a current text buffer, then applies all commands whose addresses select the buffer in sequence, then copies the buffer to standard output and clears it.

The **-n** option suppresses normal output (so that only p and w output is done). Also, some commands (n, N) do their own line reads, and some others (d, D) cause all commands following in the script to be skipped (the D command also suppresses the clearing of the current text buffer that would normally occur before the next cycle).

It is also helpful to know that there's a second buffer (called the 'hold space' that can be copied or appended to or from or swapped with the current text buffer.

An address is: a decimal numeral (which matches the line it numbers where line numbers start at 1 and run cumulatively across files), or a #$' that addresses the last line of input, or a context address, which is a #/regular expression/', in the style of ed (1) modified thus:

1. The escape sequence '\n' matches a newline embedded in the buffer, and '\t' matches a tab.
2. A command line with no addresses selects every buffer.
3. A command line with one address selects every buffer that matches that address.
4. A command line with two addresses selects the inclusive range from the first input buffer that matches the first address through the next input buffer that matches the second. (If the second address is a number less than or equal to the line number first selected, only one line is selected.) Once the second address is matched sed starts looking for the first one again; thus, any number of these ranges will be matched.

The negation operator '!' can prefix a command to apply it to every line not selected by the address(es).

In the following list of functions, the maximum number of addresses permitted for each function is indicated in parentheses.

An argument denoted "text" consists of one or more lines, with all but the last ending with '\' to hide the newline.

Backslashes in text are treated like backslashes in the replacement string of an `s' command and may be used to protect initial whitespace (blanks and tabs) against the stripping that is done on every line of the script.

An argument denoted "rfile" or "wfile" must be last on the command line. Each wfile is created before processing begins. There can be at most 10 distinct wfile arguments.

a "text" (1)
Append. Place text on output before reading the next input line.

b "label" (2)
Branch to the ':' command bearing the label. If no label is given, branch to the end of the script.

c "text" (2)
Change. Delete the current text buffer. With 0 or 1 address, or at the end of a 2-address range, place text on the output. Start the next

end of a 2 address range, place text on the output. Start the next cycle.

d (2)
Delete the current text buffer. Start the next cycle.

D (2)
Delete the first line of the current text buffer (all chars up to the first newline). Start the next cycle.

g (2)
Replace the contents of the current text buffer with the contents of the hold space.

G (2)
Append the contents of the hold space to the current text buffer.

h (2)
Copy the current text buffer into the hold space.

H (2)
Append a copy of the current text buffer to the hold space.

i "text" (1)
Insert. Place text on the standard output.

l (2)
List. Sends the pattern space to standard output. A "w" option may follow as in the s command below. Non-printable characters expand to:

\b -- backspace (ASCII 08)
\t -- tab (ASCII 09)
\n -- newline (ASCII 10)
\r -- return (ASCII 13)
\e -- escape (ASCII 27)
\xx -- the ASCII character corresponding to 2 hex digits xx.

n (2)
Copy the current text buffer to standard output. Read the next line of input into it.

N (2)
Append the next line of input to the current text buffer, inserting an embedded newline between the two. The current line number changes.

p (2)
Print. Copy the current text buffer to the standard output.

P (2)
Copy the first line of the current text buffer (all chars up to the first newline) to standard output.

q (1)
Quit. Branch to the end of the script. Do not start a new cycle.

r "rfile" (1)
Read the contents of rfile. Place them on the output before reading the next input line.

s /regular expression/replacement/flags (2)
Substitute the replacement for instances of the regular expression in the current text buffer. Any character may be used instead of #/'.

For a fuller description see ed (1).

Flags is zero or more of the following:

g -- Global. Substitute for all nonoverlapping instances of the string rather than just the first one.

p -- Print the pattern space if a replacement was made.

w -- Write. Append the current text buffer to a file argument as in a w command if a replacement is made. Standard output is used if no file argument is given

t "label" (2)
Branch-if-test. Branch to the : command with the given label if any substitutes have been made since the most recent read of an input line or execution of a 't'or 'T'. If no label is given, branch to the end of the script.

T "label" (2)
Branch-on-error. Branch to the : command with the given label if no substitutes have succeeded since the last input line or t or T command. Branch to the end of the script if no label is given.

w "wfile" (2)
Write. Append the current text buffer to wfile .

W "wfile" (2)
Write first line. Append first line of the current text buffer to wfile.

x (2)

Exchange the contents of the current text buffer and hold space.

y /string1/string2/ (2)
Translate. Replace each occurrence of a character in string1 with the corresponding character in string2. The lengths of these strings must be equal.

! "command" (2)
All-but. Apply the function (or group, if function is '{') only to lines not selected by the address(es).

: "label" (0)
This command does nothing but hold a label for 'b' and 't' commands to branch to.

= (1)
Place the current line number on the standard output as a line.

{ (2)
Execute the following commands through a matching '}' only when the current line matches the address or address range given.

An empty command is ignored.

PORTABILITY

This tool was reverse-engineered from BSD 4.1 UNIX sed, and (as far as the author's knowledge and tests can determine) is compatible with it. All documented features of BSD 4.1 sed are supported.

One undocumented feature (a leading 'n' in the first comment having the same effect as an -n command-line option) has been omitted.

The following bugs and limitations have been fixed:

- There is no hidden length limit (40 in BSD sed) on w file names.
- There is no limit (8 in BSD sed) on the length of labels.
- The exchange command now works for long pattern and hold spaces.

The following enhancements to existing commands have been made:

- a, i commands don't insist on a leading backslash-\n in the text.
- r, w commands don't insist on whitespace before the filename.
- The g, p and P options on s commands may be given in any order.

Some enhancements to regular-expression syntax have been made:

- \t is recognized in REs (and elswhere) as an escape for tab.
- In an RE, + calls for 1..n repeats of the previous pattern.

The following are completely new features:

- The l command (list, undocumented and weaker in BSD)
- The W command (write first line of pattern space to file).
- The T command (branch on last substitute failed).
- Trailing comments are now allowed on command lines.

In addition, sed's error messages have been made more specific and informative.

The implementation is also significantly smaller and faster than BSD 4.1 sed. It uses only the standard I/O library and exit.

COPYRIGHT

SEE ALSO

ed(1), grep(1), awk(1), lex(1), regexp(5)

SLB: SROFF librarian

NAME

slb - SROFF librarian

SYNOPSIS

slb - **[** [c|d|r|x|] [t[table_file]] **]** [-e] [-f] [-k] [-n] [-o] -[v] [-mmodule_file] library_file [modules...]

slb - **[** A|L|S|W|Y **]** listingfile [-U] [-v] [-mmodule_file] [modules...]

DESCRIPTION

The **slb** librarian is a tool for manipulating SROFF files (as commonly used on QDOS based systems). These are the files that are input to the linker either as individual files, or combined into a library.

The operations available in **slb** break down into two discrete functional areas:

- Library Maintenance
- SROFF Analysis

LIBRARY MAINTENANCE

These options are used to help maintain an SROFF library, and allow individual modules within the library to be manipulated. The actions you can perform are:

- **c** reate a new library;
- **d** elete modules from a library;
- **r** eplace (or add) modules to a library;
- e **x** tract modules form a library;
- **t** able the contents of a library.

ANALYSIS MODE

These options analyse the contents of SROFF files in a variety of different ways. The actions you can perform are:

- **L** ibrary order analysis.
- **A** ssembler analysis of modules **** NOT YET COMPLETE**
- **S** ROFF analysis of modules.
- Cross-Reference by XDEF name.
- Cross-Reference by File/Module name.

OPTIONS

The following options apply in all modes of operation.

-mmodulefile

modules...

Many of the options work on a list of one

or more modules. This list of module names can be provided either on the command line, or read from a file.

If inputting the names from a file specified by the **-m** module_file option then it is a simple text file with one line per entry. In most cases these are simple filenames. In special cases where the modulename is not derived from the filename then the lines are in the format filename/module.

Comments can be included in the file either by starting a line with the # symbol (in which case the whole line will be treated as a comment line) or by placing them on the same line as the module_name with a least one space before the start of the comment. Blank lines are not allowed.

The special format of **-m-** can be used to specify that the list is to be read from standard input.

-v

The verbose flag. If this is set then details of the progress of **slb** will be displayed as it runs. The details are written to standard error so that you can separate it from **-t** output using re-direction if you want to.

The following applies in library maintenance mode:

library_file

is the name of the library that is to be created or manipulated.

-c

Create the specified library. The library must not already exist unless the **-r** option is also used.

-r

Replace modules in an existing library if they already exist, and add

them otherwise. This option if used in conjunction with the **-c** option means create a new library if it does not exist, and update it otherwise.

-d

Delete modules specified from an existing library.

-x

Extract modules from an existing library. If no module list is supplied, then all modules will be extracted, and otherwise just those given in the list.

-t table_file

Table the contents of a library. If **table_file** is supplied, then the output will be written to this file, otherwise it will be written to standard output.

-e

Extension retain. Normally any extension is removed from the module name before it is inserted in the library. Thus **debug_o** and **debug_rel** would both be treated as **debug** . This flag would force them to be treated as different.

-f

File contents should be examined for module name rather than deriving it from the file name as is normally done. Only relevant to the **-c** and **-r** options. Use of this option means that you will get the correct module names even if you have renamed the files since they were compiled.

-k

Keep full filename. Normally SLB will search the filename backwards, and if it finds an underscore (other than that for the extension part of the filename) it will consider the part of the filename before the underscore to be a directory name and will remove it.

The use of the **-k** flag stops SLB removing the 'directory' part of the filename. This can be useful if you WANT to have underscores in your filenames.

-n

Causes a sequence number to be added at the front of the names of files containing modules. Affects the names of files output by the **-x** option and the names listed via the **-t** option. The module names will be of the form **nnn** _modulename where **nnn** is the sequence number in the library.

-o

***** NOT YET FULLY IMPLEMENTED *****

This option causes SLB to attempt to optimise the SROFF modules as they are put into the library. Many assemblers (such as C68) insert fields into the _o files that is not needed by the linker. This option tells SLB to remove any such references.

This optimisation mode results in the library size being smaller at the expense of taking longer to add modules to the library (because of the extra analysis that SLB has to do).

The following apply in the SROFF analysis modes.

-U

Treat Upper case as different to lower case in symbol names. The standard QDOS linker LINK, and the current C68 LD linker ignore the case of symbols.

WARNING . It is intended to reverse the meaning of this flag when the C68 LD linker is upgraded to version 2. LD version 2 will (by default) treat symbols as case significant which is more in line with standard C practice.

Currently you can only select one of the following modes in a single run. A later release of SLB is intended to raise this restriction.

-A listing_file

****** NOT YET FULLY IMPLEMENTED ******

Assembler analysis option. This produces a disassembled version of the SROFF file. The original names are used for and globally visible names, but autmatically generated names are used for any local labels (as the information regarding the original names is not held in the SROFF file).

Listing file can be specified as **-** if you wish to use standard output.

-L listingfile

Library order analysis. The linkers available under QDOS require that libraries have no backward references in them. This option is used to analyse a number of SROFF files that are to be combined into a library. It produces a file giving the dependencies amongst the various modules. This file can then be processed by the TSORT program to produce a file

order which has only forward references amongst modules.

Listing file can be specified as **-** if you wish to use standard output.

-S listing_file
SROFF analysis option. This produces an analysis of the file(s) in terms of their SROFF structure. This option is useful if you suspect that there is something wrong with the SROFF file(s). A number of validation checks are carried out during the anlysis, and if any errors are found then appropriate error messages are output.

Listing file can be specified as **-** if you wish to use standard output.

-W listing_file
Cross-Reference listing in XDEF order . This option will produce a listing in the order of Externally visible names (XDEF's) showing which module it was defined in (or not defined as the case may be). If a XDEF is defined in multiple files/modules then this will be stated as well. Following this is a list of the other files/modules that reference this item.

Listing file can be specified as **-** if you wish to use standard output.

-Y listing_file
Cross-Reference listing in File/Module order . This option will produce a listing in File/Module order showing for each module what Externally visible names (XDEF's) will be satisfied by this module, and also what External References (XREF's) are made by this module.

Listing file can be specified as **-** if you wish to use standard output.

ENVIRONMENT VARIABLES

If you are running in library maintenace mode using either the **-r** or **-d** flags (replace or delete modes) then SLB needs to use a workfile. SLB will look to see if the **TEMP** or **TMP** environment variables are set (in that order). If they are SLB will use the device named there as the location for the workfile. If neither environment variable is set then the workfile is created in the current default DATA_USE directory.

AUTHOR

David J. Walker.

22 Kimptons Mead
Potters Bar
Herts
EN6 3HZ
United Kingdom

Telephone: 0707 652791

Email address: d.j.walker@slh0101.wins.icl.co.uk

COPYRIGHT

TSORT: Topological Sort

NAME

tsort - topological sort of a directed graph

SYNOPSIS

tsort [file]

DESCRIPTION

Tsort takes a list of pairs of node names representing directed arcs in a graph and prints the nodes in topological order on standard output. Input is taken from the named file, or from standard input if no file is given.

Node names in the input are separated by white space and there must be an even number of nodes.

Presence of a node in a graph can be represented by an arc from the node to itself. This is useful when a node is not connected to any other nodes.

If the graph contains a cycle (and therefore cannot be properly sorted), one of the arcs in the cycle is ignored and the sort continues. Cycles are reported on standard error.

The most likely use that will be made of this program under QDOS is to sort the lists created by the SLB SROFF librarian utility to create valid library orderings.

TOUCH: Update File Dates

NAME

touch - update archive or modification time of file(s)

SYNOPSIS

touch [-c] [-a] [-m] [-v] filelist

DESCRIPTION

touch is used to update the modification (update) or archive (backup) times of a file. If a file does not exist, then a new zero length file is created. The time used is the current system time.

The options available with **touch** are:

-c Do not create a new file any file specified does not already exist.

-a Update the archive (backup) date of a file. If this option is specified without the **-m** option, then the modification time of existing files is left alone.

-m Update the modification time of a file. Needs to be used if you have specified the **-c** option and want both date fields to be updated.

-v Verbose flag. Causes **touch** to report on the actions it has taken.

NOTES.

1. The **-a** and **-m** options are only available on systems that have extended Tony Tebby drivers. The systems that are known to have this are the Miracle System hard disk, SMS2 systems, he QXL and the Atari ST QL Emulator with level B (or later) Tony Tebby drivers.

TO DO

Allow wildcard filenames to be used with **touch** .

Allow the date/time to be set to be specified via the command line used to invoke **touch** .

COPYRIGHT

UUE/UUD Encode/Decode Binary to ASCII

NAME

uue, uud Encode/Decode a binary file into a portable ASCII format

SYNOPSIS

uue [source-file] file-label

uud [encoded-file]

DESCRIPTION

The **uue** and **uud** programs are enhanced versions of the **uuencode** and **uudecode** programs commonly found on unix systems.

The **uue** program is used to encode a file into a format that contains only printable ASCII characters. It is typically used to enable binary files to be transmitted over electronic mail networks that only support the transfer of text. The resulting file is typically about 35% bigger than the original due to the encoding process.

The **uud** program performs the reverse operation to **uue** . It converts the encoded file from the ASCII representation produced by **uue** back to the original binary format. If any additional lines have been added to the front or end of the encoded file by the mailing process, then these extra lines are automatically stripped off by the **uud** program.

NB. The **uue** / **uud** programs do not preserve any additional information in a QDOS file header.

source-file
This is the name of the file to be encoded. If it is omitted, then it is assumed that the file will be provided as standard input.

file-label
This is the name of the file into which the binary data is to be placed when the encoded file is later decoded back into binary format.

Typically this is the same name as used for the source file, but this is not mandatory.

encoded-file

This is the name of the file which includes the encoded data. If this parameter is omitted then it is assumed that the file will be read from standard input.

The common technique is to build all the files that make up a package into an archive. In the Unix world the programs normally used for this are 'tar' or 'cpio'. This archive is then normally compressed using the 'compress' program. If the resulting file needs to be transmitted via electronic mail systems then the 'uuencode' process is applied to it.

There are a number of other progams freely available which combine the archiving/compression step. Examples of these are 'arc', 'zip', 'zoo', 'lharc'. Files produced by these programs can also use the uuenocde/uudecode process if they need to be transmitted via links that only support text file transfer.

C68 library: Indexes

INTRODUCTION

This document is intended to provide the main index into the documentation of the C68 Standard C library and the Maths library.

It provides lists of the available functions in two forms. One is a list that is organised along functional lines. The other list is an alphabetical list of all the functions. For the convenience of those who may be porting software from other systems, this latter list also contains entries for functions that are often encountered on other systems, but are not yet implemented on the QDOS/SMS C68 implementation.

COMPATIBILITY

The entries are ogranised into a number of categories as listed below. These categories are used to give an indication of how portable code that uses these routines is likely to be.

ANSI	This contains the detailed definitions for all those routines that are specified as part of the ANSI standard. Programs that are written to use just these routines can be expected to be easily portable to any system that has an ANSI C compiler. The routines that are defined as fitting into this category have fuller documentation in the LIBANSI_DOC file.
POSIX	This covers those routines that are defined as mandatory by the Posix standard Many modern operating systems (in addition to Unix based ones) will commonly support this family of calls. The routines that are defined as fitting into this category have fuller documentation in the LIBUNIX_DOC file.
XPG	This covers those routines that are optional in the POSIX standard. They are not commonly available outside the Unix environment, and even there only tend to be available on the more modern variants. The routines that are defined as fitting into this category have fuller documention in the LIBUNIX_DOC file.
UNIX	This covers routines that can be commonly found on Unix systems, but are not part of any of the formal standards mentioned above. The routines that are defined as fitting into this category have fuller documention in the LIBUNIX_DOC file.
LATTICE	This indicates that the routine is available in the Lattice family of C compilers. This reflects the fact thtat prior to C68, the main QDOS compatible C compiler was QLC which is Lattice based. Also, many of the components of C68 had their origins in the illfated PDQC prodcut that was Lattice based. The definition of Lattice has been extended to include routines commonly available in the MSDOS implementation of Lattice C. Routines that fall into this category have fuller definitions in the LIBC68_DOC file.
C68	This category refers to routines that are specific to the C68 implementation on QDOS and SMS. These are routines that do not map directly to the underlying operating system calls (which are discussed under the QDOS and SMS categories), but that do not normally exist on othr systems. They provide high level functionality that a C programmer would often want in the QDOS or SMS environments. Routines that fall into this category have fuller definitions in the LIBC68_DOC file.
QPTR	This means that the routine in question is specifically associated with the Pointer Environment. Fuller details of routines are contained in the LIBQPTR_DOC file.

The next two categories define the facilities for allowing the C programmer

The next two categories define the facilities for allowing the C programmer direct access to the underlying operating system calls. All such calls are available under two names. The QDOS category uses the original QDOS names for all such calls. The SMS category uses the newer SMS names for the calls. The functionality available is identical, so it is up to the programmer to decide whether he prefers to ue the QDOS or SMS names for such calls (or even mix them!).

QDOS — This means that the routines in question map directly onto a QDOS system call interfaces. Fuller details of such routines are contained in the LIBQDOS_DOC file.

SMS — This means that the routines in question map directly onto a SMS system interface. Fuller details of such routines are contained in the LIBSMS_DOC file.

WRITING PORTABLE PROGRAMS

The C language was designed to allow for the development of programs that would be portable across a wide variety of systems. Thus if one is careful it is possible to write programs that will compile and run unchanged on a wide variety of systems.

There have been a number of bodies that have tried to define standards that define what facilities a programmer can expect to find supported on any target platform.

The most important off these bodies is the ANSI comittee that defines the C language itself. As part of the ANSI standard, a series of library functions have been defined that a programmer can expect to be present on any system that has an ANSI C compilation system. The ANSI routines that are supported by the C68 compiler are defined in the LIBANSI_DOC file. You should find that all routines defined by ANSI are supported by C68.

The problem with the ANSI standard is that although it defines all the routines that the average application programmer might need, it does not define any lower level routines that are closer to the operating system. These lower level interfaces are often needed by systems level programmers. The POSIX standard attempts to define this more complete set of interfaces. You will find that all modern Unix compatible operating systems support the full set of POSIX calls. There are also a number of other operating systems that conform to the POSIX standard.

If you any class of routines other than those as indicated as belonging to one of the above classes, then code is unlikely to be fully portable.

REFERENCE MATERIAL

The reference books listed below were used in preparing material for inclusion in the C68 standard C library and Maths Library.

If there appeared to be any conflicting definitions, then the reference work that is nearer the top of the above list was taken as being the authoritive reference.

" **The ANSI C Standard** " by the ANSI X3J11 Technical Committee

" **The Standard C Library** " by Plauger

" **POSIX Programmers Guide** " by Donald Lewine

" **X/Open Portability Guide** " by X/Open Company, Ltd

" **SYSTEM V RELEASE 4: Programmers Reference Manual** "

" **C: A Reference Manual Edition 3** " by Harbison and Steele

[recommended as a good reference book for the average user]

" **LATTICE C Version 3: Programmers Reference manual** "

As will be seen by examination of the above list, the intention is to make the C68 implementation on QDOS and SMS ANSI and POSIX compatible as far as is realistic, while at the same time allowing those who wish to do so full access to all the capabilties of the system.

For reference on the system interfaces provided for QDOS and SMS, the following documents were used:

" **QL Technical Guide** " by David Karlin and Tony Tebby

" **QL Advanced User Guide** " by Adrian Dickens

" **QDOS/SMS Reference Manual** " as published by Jochen Merz

The pointer environment is as an optional feature with QDOS and SMSQ, while it is a standard feature of SMSQ/E and SMS2. The reference material used for the system interfaces provided to the Pointer Environment was:

" **QPTR Pointer Toolkit for the Sinclair QL** " by QJUMP

" **QPTR Pointer Environment** " as published by Jochen Merz

It is also highly recommended that you get the C68 QPTR Companion disk. This contains a Tutorial written by Tony Tebby that will help new users of the Pointer Interface get started. Thanks must go to Tony Tebby for producing this invaluable aid in the use of the Pointer Environment, and then making it freely available. Additional examples have been provided by Bob Weeks.

JOB HANDLING

Unix compatible

```
execl  execlp execv  execvp
forkl  forklp forkv  forkvp
getenv putenv rmvenv system
```

C68 compatible

QDOS compatible

```
qforkl   qforklp qforkv   qforkvp
mt_activ mt_cjob mt_frjob mt_jinf
mt_reljb mt_rjob mt_susjb mt_trapv
```

SMS compatible

```
sms_acjb sms_crjb sms_exv  sms_frjb
sms_injb sms_rmjb sms_ssjb sms_usjb
```

INPUT/OUTPUT

ANSI C compatible

```
clearerr fclose
feof     ferror  fflush  fgetc
fgetchar fgetpos fgets   fflsuhall
fopen    fopene  fprintf fputc
fread    freopen fscanf  fseek
fsetpos  ftell   fwrite  getc
getchar  gets    perror  printf
putc     putchar puts    remove
rename   rewind  scanf   setbuf
setvbuf  sprintf sscanf  tmpfile
tmpnam   ungetc
```

Posix compatible

```
chdir   close    closedir creat
dup     fcntl    fdopen   fileno
fstat   fsync    getcwd   lseek
mkdir   mkfifo   open     opendir
pclose  pipe     popen    read
readdir rewindir rmdir    seekdir
stat    telldir  tempnam  unlink
write
```

Unix compatible

```
bgets    dup2   fdmode ftruncate
iomode   mktemp opene  tell
truncate
```

Lattice C Compatible

```
clrerr (fcloseall) getch  getche
kbhit  putch       setnbf ungetch
```

C68 Compatible

```
chddir    chpdir      fgetchid  fnmatch
fqstat    fusechid    getcdd    getchid
getcname  getcpd      getfnl    opene
qdir_open qdir_delete qdir_read qdir_sort
qopen     qstat       usechid
```

QDOS compatible

```
fs_check fs_date   fs_flush fs_headr
fs_heads fs_load   fs_mdinf fs_mkdir
fs_pos   fs_posab  fs_posre fs_rename
fs_save  fs_trunc  fs_vers  fs_xinf
io_close io_delete io_edlin io_fbyte
io_fline io_format io_fstrg io_open
io_pend  io_rename io_sbyte io_sstrg
```

SMS compatible

```
ioa_cnam ioa_sown iob_elin  iob_fbyt
iob_flin iob_fmul iob_sbyt  iob_smul
iob_test iof_chek iof_date  iof_flsh
iof_load iof_minf iof_mkdir iof_posa
iof_posr iof_rhdr iof_rnam  iof_save
iof_shdr iof_trnc iof_vers  iof_xinf
```

SCREEN INPUT/OUTPUT

C68 compatible

c_extop

QDOS compatible

mt_dmode
sd_bordr sd_chenq sd_clear sd_clrbt
sd_clrln sd_clrrt sd_clrtp sd_cure
sd_curs sd_donl sd_extop sd_fill
sd_fount sd_ncol sd_nl sd_nrow
sd_pan sd_panln sd_panrt sd_pcol
sd_pixp sd_pos sd_prow sd_pxenq
sd_recol sd_scrbt sd_scrol sd_scrtp
sd_setfl sd_setin sd_setmd sd_setpa
sd_setst sd_setsz sd_setul sd_tab
sd_wdef
ut_con ut_err ut_err0 ut_mint
ut_mtext ut_scr ut_window

SMS compatible

iow_blok iow_chrq iow_clra iow_clrb
iow_clrl iow_clrr iow_clrt iow_defb
iow_defw iow_dcur iow_donl iow_ecur
iow_font iow_ncol iow_newl iow_nrow
iow_pana iow_panl iow_panr iow_pcol
iow_pixq iow_prow iow_rclr iow_scol
iow_scra iow_scrb iow_srt iow_scur
iow_sfla iow_sink iow_sova iow_spap
iow_ssiz iow_spix iow_sstr iow_sula
iow_xtop sms_dmod

GRAPHICS INPUT/OUTPUT

QDOS compatible

sd_arc sd_elipse sd_flood sd_gcur
sd_iarc sd_ielipse sd_igcur sd_iline
sd_ipoint sd_iscale sd_line sd_point
sd_scale

SMS compatible

iog_arc iog_arc_i iog_dot iog_dot_i
iog_elip iog_elip_i iog_fill iog_line
iog_line_i iog_scal iog_scal_i iog_sgcr
iog_sgcr_i

MEMORY MANAGEMENT

ANSI compatible

free calloc malloc realloc

UNIX compatible

lsbrk sbrk

LATTICE compatible

alloca (allmem) (bldmem) getmem
getml lsbrk (rbrk) rlsmem
rlsml (rstmem) sizmem

QDOS compatible

mt_alchp mt_alloc mt_alres mt_free
mt_lnkfr mt_rechp mt_reres mt_shrink
ut_link ut_unlnk

SMS compatible

mem_achp mem_alhp mem_list mem_rchp
mem_rehp mem_rlst sms_achp sms_achp_acsi
sms_alhp sms_arpa sms_frtp sms_rchp
sms_rehp sms_rrpa sms_schp

CHARACTER HANDLING

ANSI Compatible

isalnum isalpha isascii iscntrl
iscsym iscsymf isdigit isgraph
islower isprint ispunc isspace
isupper isxdigit toascii tolower
toupper

STRING PROCESSING

ANSI compatible

atof atoi atol memchr
memcmp memcpy memmove memset
strcat strchr strcoll strcmp
strcpy strcspn strerror strlen
strncat strncmp strncpy strpbrk
strrchr strspn strcpsn strstr
strtod strtok strtol strtoul
strxfrm

UNIX compatible

bcmp bcpy bzero index
memccpy rindex strpos strrpos
strcadd strccpy streadd strecpy
strfind strrspn strrstr strtrns

LATTICE Compatible

movmem setmem (stccpy) (stcd_i)
(stcd_l) (stch_i) (stch_l) (stcis)
(stcisn) (stci_d) (stci_h) (stcl_d)
(stcl_h) (stclo) stclen (stco_i)
(stco_l) (stcpm) (stcpma) stpblk
stpbrk stpchr stpchrn (stpcpy)
(stpsym) (stptok) strbpl strcmpi
strdup stricmp strins strlwr
strnicmp strnset strrev strset
strsrt strupr

C68 Compatible

itoa qstrcat qstrchr qstrcmp
qstrcpy qstricmp qstrlen qstrncat
qstrncpy qstrnicmp strfnd ut_cstr

CONVERSION

C68 compatible

cstr_to_ql d_to_qlfp i_to_qlfp l_to_qlfp
qlfp_to_d qlfp_to_f qlstr_to_c w_to_qlfp

QDOS compatible

cn_btoib cn_btoil cn_btoiw cn_date
cn_day cn_dtof cn_dtoi cn_ftod
cn_htoib cn_htoil cn_htoiw cn_itobb
cn_itobw cn_itobl cn_itod cn_itohb
cn_itohw cn_itohl

SMS compatible

cv_binib cv_binil cv_biniw cv_datil
cv_decfp cv_deciw cv_fpdec cv_hexib
cv_hexil cv_hexiw cv_ibbin cv_ibhex
cv_ilbin cv_ildat cv_ilday cv_ilhex
cv_iwbin cv_iwdec cv_iwhex

LOCALE AND LANGUAGE DEPENDENT

ANSI compatible

localeconv mblen mbtowc mbstowcs
wcstombs wctomb setlocale

SMS compatible

sms_fprm sms_lenq sms_lldm sms_lset
sms_mptr sms_pset sms_trns

DATE AND TIME

asctime ctime difftime gmtime
localtime mktime time strftime

UNIX compatible

stime tzset utime

LATTICE Compatible

jtime

QDOS compatible

mt_aclck mt_rclck mt_sclck

SMS compatible

sms_artc sms_rrtc sms_srtc

SIGNAL HANDLING

Posix Compatible

alarm kill pause raise

signal sigaction sigaddset sigdelset
sigemptyset sigfillset sigismember siglongjmp
sigpending sigprocmask sigsetjmp sigsuspend
sigset sighold sigrelse sigignore
sigpause

Unix Compatible

fraise killu raiseu

C68 compatible

sendsig set_timer_event sigcleanup

QUEUE HANDLING

QDOS compatible

io_qeof io_qin io_qout io_qset
io_qtest io_serio io_serq

SMS compatible

ioq_seof ioq_gbyt ioq_pbyt ioq_setq
ioq_test iou_ssio iou_ssq

DEVICE DRIVER LISTS

QDOS compatible

mt_ldd mt_liod mt_lpoll mt_lschd
mt_lxint mt_rdd mt_riod mt_rpoll
mt_rschd mt_rxint

SMS compatible

sms_lfsd sms_liod sms_lpol sms_lshd
sms_lexi sms_rfsd sms_riod sms_rpol
sms_rshd sms_rexi

THING ACCESS

QDOS and SMS compatible

sms_fthg sms_lthg sms_nthg sms_nthu
sms_rthg sms_uthg sms_zthg

MISCELLANEOUS

_exit abort abs access
argopt assert atexit bsearch
div exit labs ldiv
longjmp onexit qsort rand
setjmp srand wait

POSIX compatible

fpathconf getpass getpwname getpwuid
pathconf

Unix compatible

basename bufsplit chmod chown
copylist dirname endpwent getegid
geteuid getgid getopt getpid
getuid isatty link mknod
setgid setpwent setuid sleep
sync umask

LATTICE Compatible

dqsort envunpk fqsort iabs
lqsort sqsort tqsort

C68 compatible

baud beep do_sound iscon
isdevice isdirchid isdirdev isnoclose
keyrow poserr qdos1 qdos2
qdos3 qinstrn stackcheck stackreport
waitfor _CacheFlush _ProcessorType
_super _superend _user

QDOS compatible

mt_baud mt_inf mt_ipcom mt_trans

SMS compatible

sms_cach sms_comm sms_hdop sms_info
sms_iopr sms_xtop

GLOBAL VARIABLES

ANSI compatible

errno sys_errlist sys_nerr timezone
txdtn tzstn tzname

Unix compatible

UNIX compatible

environ optarg optind optopt

C68 compatible

os_nerr os_errlist
_bufsize _def_prior _endmsg _endtimeout
_memincr _memmax _memqdos _mneed
_oserr _pipesize _prog_name _stack
_stackmargin _sys_var

GLOBAL VECTORS

C68 compatible

_cmdchannels _cmdparams _cmdwildcard

BUTTON FRAME ROUTINES

bt_frame bt_free bt_prpos

WINDOW MANAGER FUNCTIONS

QPTR compatible

wm_chwin wm_clbdr wm_cluns wm_drbdr
wm_ename wm_erstr wm_findv wm_fsize
wm_idraw wm_index wm_ldraw wm_mdraw
wm_mhit wm_msect wm_pansc wm_prpos
wm_pulld wm_rname wm_rptr wm_setup
wm_stiob wm_stlob wm_swapp wm_swdef
wm_swinf wm_swlit wm_swsec wm_unset
wm_upbar wm_wdraw wm_wrset

WINDOWS MANAGER ACTION ROUTINE WRAPPERS

QPTR compatible

wm_actli wm_actme wm_drwaw wm_hitaw
wm_ctlaw

POINTER INTERFACE FUNCTIONS

QPTR compatible

iop_flim iop_lblb iop_outl iop_pick
iop_pinf iop_rptr iop_rpxl iop_rspw
iop_slnk iop_spry iop_sptr iop_svpw
iop_swdf iop_wblb iop_wrst iop_wsav
iop_wspt

STANDARD SPRITES

QPTR compatible

wm_sprite_arrow wm_sprite_cf1 wm_sprite_cf2
wm_sprite_cf3 wm_sprite_cf4 wm_sprite_f1
wm_sprite_f2 wm_sprite_f3 wm_sprite_f4
wm_sprite_f5 wm_sprite_f6 wm_sprite_f7
wm_sprite_f8 wm_sprite_f9 wm_sprite_f10
wm_sprite_hand wm_sprite_insg wm_sprite_insl
wm_sprite_left wm_sprite_move wm_sprite_null
wm_sprite_sleep wm_sprite_wake wm_sprite_zero

Alphabetical List of all Library Functions

abort	ANSI	stdlib.h	
abs	ANSI	stdlib.h	
access	POSIX	unistd.h	
acos	ANSI	math.h	
advance	XPG	regexp.h	** not implemented **
alarm	POSIX	signal.h	
allmem			
argfree	C68	sys/qlib.h	
argopt			
argpack	C68	sys/qlib.h	
argunpack	C68	sys/qlib.h	
asctime	ANSI	time.h	
asin	ANSI	math.h	
assert	ANSI	assert.h	
atan	ANSI	math.h	
atan2	ANSI	math.h	
atexit	ANSI	stdlib.h	
atof	ANSI	stdlib.h	
atoi	ANSI	stdlib.h	
atol	ANSI	stdlib.h	
basename	UNIX	libgen.h	

basename	UNIX	libgen.h	
bcmp	UNIX	memory.h	BSD Unix
bcpy	UNIX	memory.h	BSD Unix
beep	C68	sys/qlib.h	
bldmem			
bsearch	ANSI	stdlib.h	
bt_frame	QPTR		
bt_free	QPTR		
bt_prpos	QPTR		
bufsplit	UNIX	libgen.h	
bzero	UNIX	memory.h	BSD Unix
c_extop	C68	sys/qlib.h	
calloc	ANSI	stdlib.h	
ceil	ANSI	math.h	
cfgetispeed	POSIX	termios.h	** not implemented **
cfgetospeed	POSIX	termios.h	** not implemented **
cfsetispeed	POSIX	termios.h	** not implemented **
cfsetopeed	POSIX	termios.h	** not implemented **
chdir	POSIX		
chddir	C68	sys/qlib.h	
chpdir	C68	sys/qlib.h	
chmod	POSIX	sys/stat.h	
chown	POSIX	unistd.h	
chroot	POSIX	unistd.h	** not implemented **
clearerr	ANSI	stdio.h	
clock	ANSI	time.h	
close	POSIX	unistd.h	
closedir	POSIX	dirent.h	
clrerr			
cn_btolb	QDOS	qdos.h	
cn_btoil	QDOS	qdos.h	
cn_btolw	QDOS	qdos.h	
cn_date	QDOS	qdos.h	
cn_day	QDOS	qdos.h	
cn_dtof	QDOS	qdos.h	
cn_dtoi	QDOS	qdos.h	
cn_ftod	QDOS	qdos.h	
cn_htoib	QDOS	qdos.h	
cn_htoil	QDOS	qdos.h	
cn_htoiw	QDOS	qdos.h	
cn_itobb	QDOS	qdos.h	
cn_itobl	QDOS	qdos.h	
cn_itobw	QDOS	qdos.h	
cn_itod	QDOS	qdos.h	
cn_itohb	QDOS	qdos.h	
cn_itohl	QDOS	qdos.h	
cn_itojw	QDOS	qdos.h	
compile	XPG	regexp.h	** not implemented **
copylist	UNIX	libgen.h	
cos	ANSI	math.h	
cosh	ANSI	math.h	
creat	POSIX	unistd.h	
crypt	POSIX	crypt.h	** not implemented **
cstr_to_ql	C68	sys/qlib.h	
ctermid	POSIX	stdio.h	
ctime	ANSI	time.h	
cuserid	POSIX	stdio.h	
cv_binib	SMS	sms.h	
cv_binil	SMS	sms.h	
cv_biniw	SMS	sms.h	
cv_datil	SMS	sms.h	
cv_decfp	SMS	sms.h	
cv_deciw	SMS	sms.h	
cv_fpdec	SMS	sms.h	
cv_hexib	SMS	sms.h	
cv_hexil	SMS	sms.h	

cv_hexil	SMS	sms.h	
cv_hexiw	SMS	sms.h	
cv_ibbin	SMS	sms.h	
cv_ibhex	SMS	sms.h	
cv_ilbin	SMS	sms.h	
cv_ildat	SMS	sms.h	
cv_ilday	SMS	sms.h	
cv_ilhex	SMS	sms.h	
cv_iwbin	SMS	sms.h	
cv_isdec	SMS	sms.h	
cv_iwhex	SMS	sms.h	
d_to_qlfp	C68	sys/qlib.h	
difftime	ANSI	time.h	
dirname	UNIX	libgen.h	
div	ANSI	stdlib.h	
do_sound	C68	sys/qlib.h	
dqsort	LATTICE		
drand48	POSIX	stdlib.h	** not implemented **
dup	POSIX	unistd.h	
dup2	POSIX	unistd.h	
encrypt	POSIX	crypt.h	** not implemented **
endgrent	UNIX	grp.h	
endpwent	UNIX	pwd.h	
erand48	POSIX	stdlib.h	** not implemented **
erf	POSIX	math.h	** not implemented **
erfc	POSIX	math.h	** not implemented **
errno	ANSI	errno.h	
execl	POSIX	unistd.h	
execle	POSIX	unistd.h	** not implemented **
execlp	POSIX	unistd.h	
execv	POSIX	unistd.h	
execve	POSIX	unistd.h	** not implemented **
execvp	POSIX	unistd.h	
exit	ANSI	stdlib.h	
_exit	POSIX	unistd.h	
exp	ANSI	math.h	
fabs	ANSI	math.h	
fclose	ANSI	stdio.h	
fcntl	POSIX	fcntl.h	
fcvt			
fdmode			
fdopen	POSIX	stdio.h	
feof	ANSI	stdio.h	
ferror	ANSI	stdio.h	
fflush	ANSI	stdio.h	
fgetc	ANSI	stdio.h	
fgetchar			
fgetchid	C68	sys/qlib.h	
fgetpos	ANSI	stdio.h	
fgets	ANSI	stdio.h	
fileno	POSIX	stdio.h	
floor	ANSI	math.h	
fmod	ANSI	math.h	
fnmatch	C68	sys/qlib.h	
fopen	ANSI	stdio.h	
fopene			
fork	UNIX		** Not Implemented **
forkl			
forklp			
forkv			
forkvp			
fpathconf	POSIX	unistd.h	
fprintf	ANSI	stdio.h	
fputc	ANSI	stdio.h	
fputchar			
fputs	ANSI	stdio.h	

fputs	ANSI	stdio.h	
fqsort	LATTICE		
fqstat	C68	sys/qlib.h	
fread	ANSI	stdio.h	
free	ANSI	stdlib.h	
freopen	ANSI	stdio.h	
frexp	ANSI	math.h	
fscanf	ANSI	stdio.h	
fseek	ANSI	stdio.h	
fsetpos	ANSI	stdio.h	
fstat	POSIX	sys/stat.h	
fsync	XPG	unistd.h	
fs_check	QDOS	qdos.h	
fs_date	QDOS	qdos.h	
fs_flush	QDOS	qdos.h	
fs_headf	QDOS	qdos.h	
fs_geadr	QDOS	qdos.h	
fs_heads	QDOS	qdos.h	
fs_load	QDOS	qdos.h	
fs_mdinf	QDOS	qdos.h	
fs_mkdir	QDOS	qdos.h	
fs_pos	QDOS	qdos.h	
fs_posab	QDOS	qdos.h	
fs_posre	QDOS	qdos.h	
fs_rename	QDOS	qdos.h	
fs_save	QDOS	qdos.h	
fs_trunc	QDOS	qdos.h	
fs_vers	QDOS	qdos.h	
fs_xinf	QDOS	qdos.h	
ftell	ANSI	stdio.h	
ftw	POSIX	ftw.h	** not implemented **
fusechid	C68	sys/qlib.h	
fwrite	ANSI	stdio.h	
gamma	XPG	math.h	
getc	ANSI	stdio.h	
getcdd	C68	sys/qlib.h	
getchar	ANSI	stdio.h	
getchid	C68	sys/qlib.h	
getcname	C68	sys/qlib.h	
getcpd	C68	sys/qlib.h	
getcwd	POSIX	unistd.h	
getegid	POSIX	unistd.h	
getenv	ANSI	stdlib.h	
getfnl	C68	sys/qlib.h	
getmem	LATTICE		
getml	LATTICE		
geteuid	POSIX	unistd.h	
getgid	POSIX	unistd.h	
getgrent	UNIX	grp.h	
getgrgid	POSIX	grp.h	
getgrnam	POSIX	grp.h	
getgroups	POSIX	unistd.h	** not implemented **
getlogin	POSIX	unistd.h	
getopt	XPG	stdlib.h	
getpass	XPG	stdlib.h	
getpgrp	POSIX	unistd.h	** not implemented **
getpid	POSIX	unistd.h	
getppid	POSIX	unistd.h	** not implemented **
getpwent	UNIX	pwd.h	
getpwnam	POSIX	pwd.h	
getpwuid	POSIX	pwd.h	
gets	ANSI	stdio.h	
getuid	POSIX	unistd.h	
getw	XPG	stdio.h	** not implemented **
gmatch	UNIX	libgen.h	
gmtime	ANSI	time.h	

gmtime	ANSI	time.h	
hcreate	XPG	search.h	** not implemented **
hdestroy	XPG	search.h	** not implemented **
hsearch	XPG	search.h	** not implemented **
hypot	XPG	math.h	** not implemented **
i_to_qlfp	C68	sys/qlib.h	
io_close	QDOS	qdos.h	
io_delete	QDOS	qdos.h	
io_edlin	QDOS	qdos.h	
io_fbyte	QDOS	qdos.h	
io_fline	QDOS	qdos.h	
io_format	QDOS	qdos.h	
io_fstrg	QDOS	qdos.h	
io_open	QDOS	qdos.h	
io_pend	QDOS	qdos.h	
io_qeof	QDOS	qdos.h	
io_qin	QDOS	qdos.h	
io_qout	QDOS	qdos.h	
io_qset	QDOS	qdos.h	
io_qtest	QDOS	qdos.h	
io_rename	QDOS	qdos.h	
io_sbyte	QDOS	qdos.h	
io_serio	QDOS	qdos.h	
io_serq	QDOS	qdos.h	
io_sstrg	QDOS	qdos.h	
ioa_cnam	QDOS	sms.h	Only SMSQ or SMSQ/E
ioa_sown	SMS	sms.h	Only SMSQ or SMSQ/E
iob_elin	SMS	sms.h	
iob_fbyt	SMS	sms.h	
iob_flin	SMS	sms.h	
iob_fmul	SMS	sms.h	
iob_sbyt	SMS	sms.h	
iob_smul	SMS	sms.h	
iob_test	SMS	sms.h	
iof_check	SMS	sms.h	
iof_date	SMS	sms.h	
iof_flsh	SMS	sms.h	
iof_load	SMS	sms.h	
iof_minf	SMS	sms.h	
iof_mkdir	SMS	sms.h	
iof_posa	SMS	sms.h	
iof_posr	SMS	sms.h	
iof_rhdr	SMS	sms.h	
iof_rnam	SMS	sms.h	
iof_save	SMS	sms.h	
iof_shdr	SMS	sms.h	
iof_trnc	SMS	sms.h	
iof_vers	SMS	sms.h	
iof_xinf	SMS	sms.h	
iog_arc	SMS	sms.h	
iog_arc_i	SMS	sms.h	
iog_dot	SMS	sms.h	
iog_dot_i	SMS	sms.h	
iog_elip	SMS	sms.h	
iog_elip_i	SMS	sms.h	
iog_fill	SMS	sms.h	
iog_line	SMS	sms.h	
iog_line_i	SMS	sms.h	
iog_scal	SMS	sms.h	
iog_scal_i	SMS	sms.h	
iog_sgcr	SMS	sms.h	
iog_sgcr_i	SMS	sms.h	
iop_flim	QPTR	qptr.h	
iop_lblb	QPTR	qptr.h	
iop_outl	QPTR	qptr.h	
iop_pick	QPTR	qptr.h	

iop_pinf	QPTR	qptr.h
iop_rptr	QPTR	qptr.h
iop_rpxl	QPTR	qptr.h
iop_rspw	QPTR	qptr.h
iop_slnk	QPTR	qptr.h
iop_spry	QPTR	qptr.h
iop_sptr	QPTR	qptr.h
iop_svpw	QPTR	qptr.h
iop_swdef	QPTR	qptr.h
iop_wblb	QPTR	qptr.h
iop_wrst	QPTR	qptr.h
iop_wsav	QPTR	qptr.h
iop_wspt	QPTR	qptr.h
ioq_seof	SMS	sms.h
ioq_qbyt	SMS	sms.h
ioq_pbyt	SMS	sms.h
ioq_setq	SMS	sms.h
ioq_test	SMS	sms.h
iou_ssio	SMS	sms.h
iou_ssq	SMS	sms.h
iow_blok	SMS	sms.h
iow_chrq	SMS	sms.h
iow_clra	SMS	sms.h
iow_clrb	SMS	sms.h
iow_clrl	SMS	sms.h
iow_clrr	SMS	sms.h
iow_clrt	SMS	sms.h
iow_defb	SMS	sms.h
iow_defw	SMS	sms.h
iow_dcur	SMS	sms.h
iow_donl	SMS	sms.h
iow_ecur	SMS	sms.h
iow_font	SMS	sms.h
iow_ncol	SMS	sms.h
iow_newl	SMS	sms.h
iow_nrow	SMS	sms.h
iow_pana	SMS	sms.h
iow_panl	SMS	sms.h
iow_panr	SMS	sms.h
iow_pcol	SMS	sms.h
iow_pixq	SMS	sms.h
iow_prow	SMS	sms.h
iow_rclr	SMS	sms.h
iow_scol	SMS	sms.h
iow_scra	SMS	sms.h
iow_scrb	SMS	sms.h
iow_scrt	SMS	sms.h
iow_scur	SMS	sms.h
iow_sfla	SMS	sms.h
iow_sink	SMS	sms.h
iow_sova	SMS	sms.h
iow_spap	SMS	sms.h
iow_ssiz	SMS	sms.h
iow_spix	SMS	sms.h
iow_sstr	SMS	sms.h
iow_sula	SMS	sms.h
iow_xtop	SMS	sms.h
isalnum	ANSI	ctype.h
isalpha	ANSI	ctype.h
isascii	XPG	ctype.h
isatty	POSIX	unistd.h
iscntrl	ANSI	ctype.h
iscon	C68	sys/qlib.h
isdevice	C68	sys/qlib.h
isdigit	ANSI	ctype.h

isdirchid	C68	sys/qlib.h	
isdirdev	C68	sys/qlib.h	
isgraph	ANSI	ctype.h	
islower	ANSI	ctype.h	
isnan	XPG	math.h	** not implemented **
isnoclose	C68	sys/qlib.h	
isprint	ANSI	ctype.h	
ispunct	ANSI	ctype.h	
isspace	ANSI	ctype.h	
isupper	ANSI	ctype.h	
isxdigit	ANSI	ctype.h	
l_to_qlfp	C68	sys/qdos.h	
j0	XPG	math.h	** not implemented **
j1	XPG	math.h	** not implemented **
jn	XPG	math.h	** not implemented **
jrand48	POSIX	stdlib.h	** not implemented **
kill	POSIX	signal.h	
labs	ANSI	stdlib.h	
lcong48	POSIX	stdlib.h	** not implemented **
ldexp	ANSI	math.h	
ldiv	ANSI	stdlib.h	
lfind	XPG	search.h	** not implemented **
lgamma	XPG	math.h	** not implemented **
link	POSIX	unistd.h	
localeconv	ANSI	locale.h	
localtime	ANSI	time.h	
log	ANSI	math.h	
log10	ANSI	math.h	
longjmp	ANSI	setjmp.h	
lrand48	POSIX	stdlib.h	** not implemented **
lsearch	XPG	search.h	** not implemented **
lseek	POSIX	unistd.h	
malloc	ANSI	stdlib.h	
mblen	ANSI	stdlib.h	
mbstowcs	ANSI	stdlib.h	
mbtowc	ANSI	stdlib.h	
memccpy	XPG	string.h	
memchr	ANSI	string.h	
memcmp	ANSI	string.h	
memcpy	ANSI	string.h	
memmove	ANSI	string.h	
memset	ANSI	string.h	
mem_achp	SMS	sms.h	
mem_alhp	SMS	sms.h	
mem_rchp	SMS	sms.h	
mem_rehp	SMS	sms.h	
mem_rlst	SMS	sms.h	
mkdir	POSIX	sys/stat.h	
mkfifo	POSIX	sys/stat.h	
mknod	UNIX	sys/stat.h	
mktime	ANSI	time.h	
modf	ANSI	math.h	
modff	UNIX	math.h	
mrand48	POSIX	stdlib.h	** not implemented **
msgctl	XPG	sys/ipc.h	** not implemented **
		sys/msg.h	
msgget	XPG	sys/ipc.h	** not implemented **
		sys/msg.h	
msgrcv	XPG	sys/ipc.h	** not implemented **
		sys/msg.h	
msgsnd	XPG	sys/ipc.h	** not implemented **
		sys/msg.h	
mt_aclck	QDOS	qdos.h	
mt_activ	QDOS	qdos.h	
mt_alchp	QDOS	qdos.h	

mt_alloc	QDOS	qdos.h	
mt_alres	QDOS	qdos.h	
mt_baud	QDOS	qdos.h	
mt_cjob	QDOS	qdos.h	
mt_dmode	QDOS	qdos.h	
mt_free	QDOS	qdos.h	
mt_frjob	QDOS	qdos.h	
mt_inf	QDOS	qdos.h	
mt_ipcom	QDOS	qdos.h	
mt_jinf	QDOS	qdos.h	
mt_ldd	QDOS	qdos.h	link directory driver
mt_liod	QDOS	qdos.h	link I/O driver
mt_lnkfr	QDOS	qdos.h	
mt_lpoll	QDOS	qdos.h	link polled loop
mt_lschd	QDOS	qdos.h	link scheduler loop
mt_lxint	QDOS	qdos.h	
mt_rclck	QDOS	qdos.h	
mt_rechp	QDOS	qdos.h	release common heap
mt_reljb	QDOS	qdos.h	
mt_reres	QDOS	qdos.h	
mt_riod	QDOS	qdos.h	
mt_rjob	QDOS	qdos.h	
mt_rpoll	QDOS	qdos.h	
mt_rschd	QDOS	qdos.h	
mt_rxint	QDOS	qdos.h	
mt_sclck	QDOS	qdos.h	
mt_shrink	QDOS	qdos.h	
mt_susjb	QDOS	qdos.h	
mt_trans	QDOS	qdos.h	
mt_trapv	QDOS	qdos.h	
nice.h	XPG	unistd.h	** not implemented **
nl_langinfo	XPG	langinfo.h	** not implemented **
nrand48	POSIX	stdlib.h	** not implemented **
open	POSIX	fcntl.h	
		sys/stat.h	
open_qdir	C68	sys/qlib.h	
opendir	POSIX	dirent.h	
pathconf	POSIX	unistd.h	
pause	POSIX	unistd.h	
pclose	XPG	stdio.h	
perror	ANSI	stdio.h	
pipe	POSIX	unistd.h	
popen	XPG	stdio.h	
poserr	C68	sys/qlib.h	
pow	ANSI	math.h	
printf	ANSI	stdio.h	
putc	ANSI	stdio.h	
putchar	ANSI	stdio.h	
putenv	XPG	unistd.h	
puts	ANSI	stdio.h	
putw	XPG	stdio.h	** not implemented **
qdir_delete	C68	sys/qlib.h	
qdir_read	C68	sys/qlib.h	
qdir_sort	C68	sys/qlib.h	
qdos1	LATTICE	sys/qlib.h	
qdos2	LATTICE	sys/qlib.h	
qdos3	LATTICE	sys/qlib.h	
qforkl	C68	sys/qlib.h	
qforklp	C68	sys/qlib.h	
qforkv	C68	sys/qlib.h	
qforkvp	C68	sys/qlib.h	
qinstrn	C68	sys/qlib.h	
qlfp_to_d	C68	sys/qlib.h	
qlfp_to_f	C68	sys/qlib.h	
qopen	C68	fcntl.h	

- -

qsort	ANSI	stdlib.h
qstat	C68	sys/qlib.h
qstrcat	C68	sys/qlib.h
qstrchr	C68	sys/qlib.h
qstrcmp	C68	sys/qlib.h
qstrcpy	C68	sys/qlib.h
qstricmp	C68	sys/qlib.h
qstrlen	C68	sys/qlib.h
qstrncat	C68	sys/qlib.h
qstrncmp	C68	sys/qlib.h
qstrncpy	C68	sys/qlib.h
qstrnicmp	C68	sys/qlib.h
raise	ANSI	signal.h
rand	ANSI	stdlib.h
read	POSIX	unistd.h
read_qdir	C68	sys/qlib.h
readdir	POSIX	dirent.h
realloc	ANSI	stdlib.h
remove	ANSI	stdio.h
rename	ANSI	stdio.h
rewind	ANSI	stdio.h
rewinddir	POSIX	dirent.h
rmdir	POSIX	unistd.h
scanf	ANSI	stdio.h
sd_arc	QDOS	qdos.h
sd_bordr	QDOS	qdos.h
sd_chenq	QDOS	qdos.h
sd_clear	QDOS	qdos.h
sd_clrbt	QDOS	qdos.h
sd_clrln	QDOS	qdos.h
sd_clrrt	QDOS	qdos.h
sd_clrtp	QDOS	qdos.h
sd_cure	QDOS	qdos.h
sd_curs	QDOS	qdos.h
sd_donl	QDOS	qdos.h
sd_elipse	QDOS	qdos.h
sd_extop	QDOS	qdos.h
sd_fill	QDOS	qdos.h
sd_flood	QDOS	qdos.h
sd_fount	QDOS	qdos.h
sd_gcur	QDOS	qdos.h
sd_iarc	QDOS	qdos.h
sd_ielipse	QDOS	qdos.h
sd_igcur	QDOS	qdos.h
sd_iline	QDOS	qdos.h
sd_ipoint	QDOS	qdos.h
sd_iscale	QDOS	qdos.h
sd_line	QDOS	qdos.h
sd_ncol	QDOS	qdos.h
sd_nl	QDOS	qdos.h
sd_nrow	QDOS	qdos.h
sd_pan	QDOS	qdos.h
sd_panln	QDOS	qdos.h
sd_panrt	QDOS	qdos.h
sd_pcol	QDOS	qdos.h
sd_pixp	QDOS	qdos.h
sd_point	QDOS	qdos.h
sd_pos	QDOS	qdos.h
sd_prow	QDOS	qdos.h
sd_pxenq	QDOS	qdos.h
sd_recol	QDOS	qdos.h
sd_scale	QDOS	qdos.h
sd_scrbt	QDOS	qdos.h
sd_scrol	QDOS	qdos.h
sd_scrtp	QDOS	qdos.h

sd_setfl	QDOS	qdos.h	
sd_setin	QDOS	qdos.h	
sd_setmd	QDOS	qdos.h	
sd_setpa	QDOS	qdos.h	
sd_setst	QDOS	qdos.h	
sd_setsz	QDOS	qdos.h	
sd_setul	QDOS	qdos.h	
sd_tab	QDOS	qdos.h	
sd_wdef	QDOS	qdos.h	
seed48	POSIX	stdlib.h	** not implemented **
seekdir	XPG	dirent.h	
semctl	XPG	sys/ipc.h sys/sem.h	** not implemented **
semget	XPG	sys/ipc.h sys/sem.h	** not implemented **
semop	XPG	sys/ipc.h sys/sem.h	** not implemented **
sendsig	C68	sys/signal.h	
setbuf	ANSI	stdio.h	
seteuid	UNIX	unistd.h	
setgid	POSIX	unistd.h	
setgrent	UNIX	grp.h	
setjmp	ANSI	setjmp.h	
setkey	XPG	crypt.h	** not implemented **
setlocale	ANSI	locale.h	
setpgid	POSIX	unistd.h	
setpwent	UNIX	pwd.h	
setsid	POSIX	unistd.h	** not implemented **
setuid	POSIX	unistd.h	
setvbuf	ANSI	stdio.h	
set_timer_event	C68	sys/signal.h	
shmat	XPG	sys/ipc.h sys/shm.h	** not implemented **
shmctl	XPG	sys/ipc.h sys/shm.h	** not implemented **
shmdt	XPG	sys/ipc.h sys/shm.h	** not implemented **
shmget	XPG	sys/ipc.h sys/shm.h	** not implemented **
sigaction	POSIX	signal.h	
sigaddset	POSIX	signal.h	
sigcleanup	C68	sys/signal.h	
sigdelset	POSIX	signal.h	
sigemptyset	POSIX	signal.h	
sigfillset	POSIX	signal.h	
sigismember	POSIX	signal.h	
siglongjmp	POSIX	signal.h	
signal	ANSI	signal.h	
sigpending	POSIX	signal.h	
sigprocmask	POSIX	signal.h	
sigsetjmp	POSIX	setjmp.h	
sigsuspend	POSIX	signal.h	
sin	ANSI	math.h	
sinh	ANSI	math.h	
sleep	POSIX	unstd.h	
sprintf	ANSI	stdio.h	
sqrt	ANSI	math.h	
sms_achp	SMS	sms.h	
sms_acjb	SMS	sms.h	
sms_alhp	SMS	sms.h	
sns_arpa	SMS	sms.h	
sms_artc	SMS	sms.h	
sms_cach	SMS	sms.h	Not SMS/2
sms_comm	SMS	sms.h	
sms_crjb	SMS	sms.h	

sms_fprm	SMS	sms.h	Not SMS/2
sms_frjb	SMS	sms.h	
sms_frtp	SMS	sms.h	
sms_fthg	QDOS	qdos.h	
sms_fthg	SMS	sms.h	
sms_hdop	SMS	sms.h	
sms_info	SMS	sms.h	
sms_injb	SMS	sms.h	
sms_iopr	SMS	sms.h	Not SMS/2
sms_lenq	SMS	sms.h	Not SMS/2
sms_lexi	SMS	sms.h	
sms_lfsd	SMS	sms.h	
sms_liod	SMS	sms.h	
sms_lldm	SMS	sms.h	Not SMS/2
sms_lpol	SMS	sms.h	
sms_lset	SMS	sms.h	Not SMS/2
sms_lshd	SMS	sms.h	
sms_lthg	QDOS	qdos.h	
sms_lthg	SMS	sms.h	
sms_mptr	SMS	sms.h	Not SMS/2
sms_nthg	QDOS	qdos.h	Next thing
sms_nthg	SMS	sms.h	Next thing
sms_nthu	QDOS	qdos.h	Next thing user
sms_nthu	SMS	sms.h	Next thing user
sms_pset	SMS	sms.h	Not SMS/2
sms_rchp	SMS	sms.h	
sms_rehp	SMS	sms.h	
sms_rexi	SMS	sms.h	
sms_rfsd	SMS	sms.h	
sms_riod	SMS	sms.h	
sms_rmjb	SMS	sms.h	
sms_rpol	SMS	sms.h	
sms_rrpa	SMS	sms.h	Release res. proc. area
sms_rrtc	SMS	sms.h	Read real time clock
sms_rshd	SMS	sms.h	
sms_rthg	QDOS	qdos.h	
sms_rthg	SMS	sms.h	
sms_schp	SMS	sms.h	Not SMS/2
sms_srtc	SMS	sms.h	Set real time clock
sms_ssjb	SMS	sms.h	
sms_trns	SMS	sms.h	Set translate table(s)
sms_usjb	SMS	sms.h	
sms_uthg	QDOS	qdos.h	
sms_uthg	SMS	sms.h	
sms_xtop	SMS	sms.h	Not SMS/2
sms_zthg	QDOS	qdos.h	
sms_zthg	SMS	sms.h	
srand	ANSI	stdlib.h	
srand48	POSIX	stdlib.h	** not implemented **
sscanf	ANSI	stdio.h	
stackcheck	C68	sys/qlib.h	
stackreport	C68	sys/qlib.h	
stat	POSIX	sys/stat.h	
step	XPG	regexp.h	** not implemented **
strcat	ANSI	string.h	
strcadd	UNIX	libgen.h	Part of LIBGEN library
strccpy	UNIX	libgen.h	
strchr	ANSI	string.h	
strcmp	ANSI	string.h	
strcoll	ANSI	string.h	
strcpy	ANSI	string.h	
strcspn	ANSI	string.h	
streadd	UNIX	libgen.h	
strecpy	UNIX	libgen.h	
strerror	ANSI	string.h	

```
strfind             UNIX    libgen.h
strfnd              C68     string.h
strftime            ANSI    time.h
stricmp             UNIX    string.h
strlen              ANSI    string.h
strmfe              LATTICE sys/qlib.h
strmfn              LATTICE sys/qlib.h
strmfp              LATTICE sys/qlib.h
strncat             ANSI    string.h
strncmp             ANSI    string.h
strncpy             ANSI    string.h
strnicmp            UNIX    string.h
strpbrk             ANSI    string.h
strrchr             ANSI    string.h
strrspn             UNIX    libgen.h
strrstr             UNIX    string.h
strspn              ANSI    string.h
strstr              ANSI    string.h
strtod              ANSI    stdlib.h
strtok              ANSI    string.h
strtol              ANSI    stdlib.h
strtoul             ANSI    stdlib.h
strtrns             UNIX    libgen.h
strxfrm             ANSI    string.h
swab                XPG     stdlib.h     ** not implemented **
sysconf             POSIX   unstd.h      ** not implemented **
system              ANSI    stdlib.h
tan                 ANSI    math.h
tanh                ANSI    math.h
tcdrain             POSIX   termios.h    ** not implemented **
tcflow              POSIX   termios.h    ** not implemented **
tcflush             POSIX   termios.h    ** not implemented **
tcgetattr           POSIX   termios.h    ** not implemented **
tcgetpgrp           POSIX   termios.h    ** not implemented **
tcsendbreak         POSIX   termios.h    ** not implemented **
tcsetattr           POSIX   termios.h    ** not implemented **
tcsetpgrp           POSIX   termios.h    ** not implemented **
tdelete             XPG     search.h     ** not implemented **
telldir             XPG     dirent.h
tempnam             XPG     stdio.h
tfind               XPG     search.h     ** not implemented **
time                ANSI    time.h
times               POSIX   sys/times.h
tmpfile             ANSI    stdio.h
tmpnam              ANSI    stdio.h
toascii             XPG     ctype.h
tolower             ANSI    ctype.h
_tolower            XPG     ctype.h
toupper             ANSI    ctype.h
_toupper            XPG     ctype.h
tsearch             XPG     search.h     ** not implemented **
ttyname             POSIX   unistd.h
twalk               XPG     search.h     ** not implemented **
tzset               POSIX   time.h
ulimit              XPG     ulimit.h     ** not implemented **
umask               POSIX   sys/stat.h
uname               POSIX   sys/utsname  ** not implemented **
ungetc              ANSI    stdio.h
unlink              POSIX   unsitd.h
usechid             C68     sys/qlib.h
utime               POSIX   utime.h      ** not implemented **
ut_con              QDOS    qdos.h
ut_err              QDOS    qdos.h
ut_err0             QDOS    qdos.h
ut_link             QDOS    qdos.h
```

ut_mint	QDOS	qdos.h	
ut_mtext	QDOS	qdos.h	
ut_scr	QDOS	qdos.h	
ut_unlnk	QDOS	qdos.h	
ut_window	QDOS	qdos.h	
va_arg	ANSI	stdarg.h	
va_end	ANSI	stdarg.h	
va_list	ANSI	stdarg.h	
va_start	ANSI	stdarg.h	
vfprintf	ANSI	stdio.h	
vprintf	ANSI	stdio.h	
vsprintf	ANSI	stdio.h	
w_to_qlfp	C68	sys/qlib.h	
wait	POSIX	sys/wait.h	
waitfor	C68	sys/qlib.h	
waitpid	POSIX	sys/wait.h	
wcstombs	ANSI	stdlib.h	
wctomb	ANSI	stdlib.h	
wm_chwin	QPTR	qptr.h	change event handling
wm_clbdr	QPTR	qptr.h	
wm_cluns	QPTR	qptr.h	
wm_drbdr	QPTR	qptr.h	draw border arount item
wm_ename	QPTR	qptr.h	Edit name
wm_erstr	QPTR	qptr.h	Get string for error code
wm_findv	QPTR	qptr.h	
wm_fsize	QPTR	qptr.h	Find size of layout
wm_ldraw	QPTR	qptr.h	Draw info sub-windows
wm_index	QPTR	qptr.h	standard sub-window index
wm_ldraw	QPTR	qptr.h	loose menu item drawing
wm_mdraw	QPTR	qptr.h	standard menu drawing
wm_mhit	QPTR	qptr.h	Window hit routine
wm_msect	QPTR	qptr.h	Find menu section
wm_pansc	QPTR	qptr.h	Pan/scroll standard menu
wm_prpos	QPTR	qptr.h	primary window position
wm_pulld	QPTR	qptr.h	pull down window open
wm_rname	QPTR	qptr.h	Read name
wm_rptr	QPTR	qptr.h	Read pointer
wm_setup	QPTR	qptr.h	setup managed window
wm_sprite_arrow	QPTR	qptr.h	
wm_sprite_cf1	QPTR	qptr.h	
wm_sprite_cf2	QPTR	qptr.h	
wm_sprite_cf3	QPTR	qptr.h	
wm_sprite_cf4	QPTR	qptr.h	
wm_sprite_f1	QPTR	qptr.h	
wm_sprite_f2	QPTR	qptr.h	
wm_sprite_f3	QPTR	qptr.h	
wm_sprite_f4	QPTR	qptr.h	
wm_sprite_f5	QPTR	qptr.h	
wm_sprite_f6	QPTR	qptr.h	
wm_sprite_f7	QPTR	qptr.h	
wm_sprite_f8	QPTR	qptr.h	
wm_sprite_f9	QPTR	qptr.h	
wm_sprite_f10	QPTR	qptr.h	
wm_sprite_hand	QPTR	qptr.h	
wm_sprite_insg	QPTR	qptr.h	
wm_sprite_insl	QPTR	qptr.h	
wm_sprite_left	QPTR	qptr.h	
wm_sprite_move	QPTR	qptr.h	
wm_sprite_null	QPTR	qptr.h	
wm_sprite_sleep	QPTR	qptr.h	
wm_sprite_wake	QPTR	qptr.h	
wm_sprite_zero	QPTR	qptr.h	
wm_stiob	QPTR	qptr.h	Set information object
wm_stlob	QPTR	qptr.h	Set loose item object
wm_swapp	QPTR	qptr.h	Set window to app sub-wind

wm_swdef	QPTR	qptr.h	set sub-window definition
wm_swinf	QPTR	qptr.h	set window to info window
wm_swlit	QPTR	qptr.h	set window to loose item
wm_swsec	QPTR	qptr.h	set to app sub-win section
wm_unset	QPTR	qptr.h	window unset
wm_upbar	QPTR	qptr.h	update pan/scroll bars
wm_wdraw	QPTR	qptr.h	draw window contents
wm_wrset	QPTR	qptr.h	window reset
write	POSIX	unistd.h	
y0	XPG	math.h	** not implemented **
y1	XPG	math.h	** not implemented **
yn	XPG	math.h	** not implemented **
_chkufbs	C68		
_CacheFlush	C68	sys/qlib.h	
_ProcessorType	C68	sys/qlib.h	
_super	C68	sys/qlib.h	
_superend	C68	sys/qlib.h	
_user	C68	sys/qlib.h	

AMENDMENT HISTORY

29 Aug 94	New document that brings together in one place summary information covering all aspects of the LIBC_A library.
25 Mar 95	Added new traps specific to SMSQ and SMSQ-E.
07 Dec 96	Added routines that are in the libgen.h header file (and thus the LIBGEN_A library).
12 Mar 98	Added strrstr() routine.
16 May 98	Added _CacheFlush() and _ProcessorType().

Standard C library: ANSI routines

This section of the C68 library documentation covers those routines in the C68 Standard C library that provide ANSI compatibility, except for any ANSI routines that use the math.h header file (which are covered in the LIBM_DOC file).

---

int abort(void)

Defined in stdlib.h

---

int abs(int value)

Compute absolute value of integer value.

Defined in stdlib.h

---

char *asctime(struct tm *t)

Convert time of 'struct tm' type (defined in time.h) to a time string of the format:

```
Fri Sep 13 00:00:00 1986\n\0
```

Time zone correction is done if required.

Defined in time.h

---

void assert(x)

Test if specified condition is true, and if not output a message indicating the name and line number of the original source file of the statement that failed, plus the text of the failed test. This is actually a macro, and will only generate code if the NDEBUG preprocessor constant is defined.

Macro defined in assert.h

---

void atexit (void (*function)(void))

Register a function to be called during exit processing.

Defined in stdlib.h

---

double atof (const char *)

Convert an ASCII string into a floating point number.

Defined in stdlib.h

---

int atoi(const char *s)

Convert a string to a integer using base 10 arithmetic. If you want to use a base other than 10, you must use strtol().

Defined in stdlib.h

---

long atol(const char *s)

Convert a string to a long integer using base 10 arithmetic. If you want to use a base other than 10, then use strtol().

Defined in stdlib.h

---

void *bsearch (void * key, void * base, size_t count,
size_t size, int (*cmpfunction)())

Search an array of objects pointed to by base for an element that matches key using the supplied comparison function.

Defined in stdlib.h

---

char *calloc(size_t nelt, size_t esize)

Allocate enough memory to hold an array of objects of the specifed size and number. Initialise all bits to zero.

Defined in stdlib.h

---

void clearerr(FILE *fp)

Clear any error or EOF indicator for a file.

Defined in stdio.h

---

char *ctime(const time_t *t)

Convert a 'time_t' time into an ASCII string which will be of the form:

```
Fri Sep 13 00:00:00 1986\n\0
```

Timezone corrections are made if required.

Defined in time.h

---

double difftime(time_t time1, time_t time2)

Calculate the difference between two times in seconds.

Defined in time.h

---

div_t div(int numer, int denom)

Calculate the quotient and remainder of a number. The answer is returned in the structure div_t.

Defined in stdlib.h

---

void exit(int code)

Exit a program normally returning given code as exit status.

Defined in stdlib.h

---

int fclose(FILE *fp)

Close a file.

Defined in stdio.h

---

int feof(FILE *fp)

Test for end of file condition.

Defined in stdio.h

---

int ferror(FILE *fp)

Get the last error code of an I/O stream..

Defined in stdio.h

---

int fflush(FILE *fp)

Force the C level buffers associated with a stream to be flushed. If the parameter is NULL, then this means flush all streams.

Defined in stdio.h

---

int fgetc(FILE *fp)

Get a character from a stream.

Defined in stdio.h

---

int fgetpos(FILE * fp, fpos_t * pos)

Store current file position in object pointed to by pos.
Defined in stdio.h

---

char * fgets(char *buf, int length, FILE *fp)
Read a string from a stream.
Defined in stdio.h

FILE * fopen (char * name, char * mode)
Open a file.
Defined in stdio.h

---

int fprintf(FILE *fp, char *s, ...)
Formatted output to a specified file
Defined in stdio.h

---

int fputc(int c, FILE *fp)
Defined in stdio.h

---

int fputs(char *s, FILE *fp)
Defined in stdio.h

---

int fread(char *buf, size_t bsize, size_t n, FILE *fp)
Read unformatted data from a specified file
Defined in stdio.h

---

int free(char *s)
Free memory that was previously allocated using malloc().
Defined in stdlib.h

FILE *freopen(char *name, char *mode, FILE *fp)
Defined in stdio.h

---

int fscanf(FILE *fp, char *s, ...)
Formatted input from a specified file.
Defined in stdio.h

---

int fseek(FILE *fp, long rpos, int mode)
Set file position. Mode defines what the position is to be relative to.
Defined in stdio.h

---

int fsetpos (FILE *fp, fpos_t * pos)
Set file position according to values in the object pointed to by pos.
Defined in stdio.h

---

long ftell(FILE *fp)
Return the current file position for the specified file.
Defined in stdio.h

---

int fwrite(char *buf, size_t bsize, size_t n, FILE *fp)
Output unformatted data to a specified file.
Defined in stdio.h

---

int getc(FILE *fp)
Read a character from the specified stream. This is normally a macro version of fgetc().
Defined in stdio.h

---

int getchar(void)
Read a character from the stdin. Macro
Defined in stdio.h

---

char *getenv (char *name)

Search the environment for a string of the value name=value, and if it exists return a pointer to the value part of the string. If it does not exist, then return NULL.

Defined in stdlib.h

---

char *gets(char *buf)

Defined in stdio.h

---

struct tm *gmtime(long *t)

Convert a raw time into a tm type structure.

Defined in time.h

---

int isalnum(int c)

Check that a character is alphanumeric. Casre must be taken that a negative value is not passed to this routine, or the results can be unpredictable.

---

int isalpha(int c)

Check that a character is alphabetic.

Defined in ctype.h

---

int iscntrl(int c)

Defined in ctype.h

---

int isdigit(int c)

Defined in ctype.h

---

int isgraph(int c)

Defined in ctype.h

---

int islower(int c)

Defined in ctype.h

---

int isprint(int c)

Defined in ctype.h

---

int ispunct(int c)

Defined in ctype.h

---

int isspace(int c)

Check if a character is a whitespace character. As well as the obvious space character, this also includes tabas and newline.

Defined in ctype.h

---

int isupper(int c)

Defined in ctype.h

---

int isxdigit(int c)

Macros in ctype.h (or function if ctype.h not included)

Defined in ctype.h

---

long labs(long i)

Computes the absolute value of an long integer.

Defined in stdlib.h

---

ldiv_t ldiv(long numer, long denom)

ANSI compatible routine to calculate the quotient and remainder of a number. The answer is returned in the structure div_t.

Defined in stdlib.h

---

struct tm *localtime(long *t)

Defined in time.h

---

void longjmp(jmp_buf *save, int val)

Defined in setjmp.h

---

int main(int argc, char *argv[], char *envp[])

Start function in a users program.

Defined in stdlib.h

---

char *malloc(int size)

Allocate memory.

Defined in stdlib.h

---

int mblen(const char *s, size_t n)

ANSI compatible routine to determine the number of bytes in the multi-byte character pointed to by s.

Defined in stdlib.h

---

size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n)

ANSI compatible routine to convert a multi-byte character strings to a wide character string.

Defined in stdlib.h

---

int mbtowc(wchar_t *pwc, const char *s, size_t n)

ANSI compatible routine to convert a multi-byte character to a wide character.

Defined ins tdlib.h

---

char *memchr(cconst void *a, int c, size_t length)

Search memory for a character.

Defined in string.h

---

int memcmp(const void *a, const void *b, size_t length)

Compare two memory areas.

Defined in string.h

---

char *memcpy(void *to, const void *from, size_t length)

Move memory as fast as possible). Not safe if areas overlap.

Defined in string.h

---

void memmove(void *dest, const void *source, size_t length)

Move memory safely even if areas overlap.

Defined in string.h

---

char *memset(void *to, int c, size_t length)

Defined in string.h

---

time_t mktime(struct tm * tmptr)

ANSI compatible routine to convert a time between the struct tm' format and the raw 'time_t' format.

Defined in time.h

---

int perror(char *prompt)

Print the text corresponding to the error code held in the global variable 'errno'. If the 'prompt' parameter is not NULL, then put this at the front of the error mesage. This routine only understands the error codes defined in errno.h (for QDOS error codes refer to the routine 'poserr' in the QDOS specific part of the library).

Defined in errno.h

---

int printf(char *fmt, ...)

Print formatted output to stdout.

Defined in stdio.h

---

int putc(int c, FILE *fp)

Send character to a file.

Macro defined in stdio.h

---

int putchar(int c)

Send character to stdout

Macro defined in stdio.h

---

int putenv(char * string)

Set environment variable

Defined in stdlib.h

---

int puts(char *s)

Send string to stdout.

Defined in stdio.h

---

void qsort(char *a, int n, int size, int (*cmp_func)())

The qsort function sorts an array of n objects, the initial element of which is pointed to by 'base'. The size of each object is specified by 'size'.

The contents of this array are sorted into ascending order according to a supplied comparison function 'cmp_func' which is called with two arguments that point to the objects being compared. The function must return an integer less than, equal to, or greater than zero if the first argument is considered to be repectively less than, equal to, or greater than the second.

If two elements compare as equal then their order in the array is unspecified.

Defined in stdlib.h

Note The LIBC68_doc file defines a number of variants of qsort for sorting arrays of standard data types. These routines are:

dqsort	sort an array of doubles
fqsort	sort an array of floats
lqsort	sort an array of long integers
sqsort	sort an array of short integers
tqsort	sort an array of text pointers

These variants of qsort() are not defined by the ANSI C standard, so may not always be found on other platforms.

---

int rand(void)

Generate a pseudo-random number. It will be in range 0 to RAND_MAX (defined in stdlib.h). See also srand().

Defined in stdlib.h

---

char *realloc(char *old_memory, unsigned new_size)

Defined in stdlib.h

---

int remove(char *name);

Delete a file. 0 if ok, -1 if error.

Defined in stdio.h

---

int rename(char *old_name, char *new_name)

Routine to rename a file. Both old and new names must be on same device (if no device given default directory is used). Returns 0 on success, -1 on failure. Will not work on standard QL systems without either Toolkit 2 or a disk interface present.

Defined in stdio.h

---

int rewind(FILE *fp)

Defined in stdio.h

---

int scanf(char *fmt, ...)

Defined in stdio.h

---

void setbuf(FILE *fp, char *buf)

Supply buffer to be used for a stream.

Defined in stdio.h

---

int setjmp (jmp_buf * save_area)

Defined in setjmp.h

---

char * setlocale (int category, const char *locale)

ANSI compatible routine to modify or query a programs locale.

Defined in locale.h

---

int setvbuf(FILE *fp, char *buf, int type, int size)

Set buffering strategy details for a stream.

Defined in stdio.h

---

int sprintf(char *str, char *fmt, ...)

Defined in stdio.h

---

void srand (unsigned int seed)

Seed the pseudo-random number generator. If this function is not called, then the sequence of numbers returned by rand( ) will be equivalent to setting the seed to 1.

Defined in stdlib.h

---

int sscanf(char *str, char *fmt, ...)

Defined in stdio.h

---

char *strcat(char *dest, const char *src)

Concatenate two strings. Returns pointer to the resultant string.

Defined in string.h

---

char *strchr(const char *s, int c)

Search a string for a specified character

Defined in string.h

---

int strcmp(const char *a, const char *b)

Compare two strings.

Defined in string.h

---

int strcoll(const char *, const char*)

Locale-specific string compare. In C68 this is a dummy and is functionally identical to strcmp().

Defined in string.h

---

char *strcpy(char *to, const char *from)

Defined in string.h

---

int strcspn(const char *a, const char *b)

Defined in string.h

---

char *strerror(int)

Defined in string.h

---

size_t strftime(char *s, size_t maxsize, const char *format,

const struct tm *timeptr)

ANSI compatible routine to convert a 'struct tm' time into an ASCII string controlled by a format conversion string.

The list of conversion options includes extensions specified by various Unix variants, and by POSIX. The full list of conversions supported is:

 %a ANSI abbreviated weekday name e.g. 'Mon'
 %A ANSI full weekday name e.g. 'Monday'
 %b ANSI abbreviated month name e.g. 'Feb'
 %B ANSI full month name e.g. 'February'
 %c ANSI local-specific date and time
 %d ANSI day of month as decimal integer (01-31)
 %D POSIX date in the form %m/%d/%y
 %e UNIX day of month (1-31)

%h POSIX locale's abreviated month name
%H ANSI the hour (24 hour clock) (00-23)
%I ANSI the hour (12 hour clock) (01-12)
%j ANSI day of year as decimal number (001-366)
%m ANSI month as decimal number (01-12)
%M ANSI minute as decimal number (00-59)
%n POSIX newline character
%p ANSI locale's equivalent of AM/PM
%r POSIX time in %I:%M:%S AM|PM format
%R UNIX time as %H:%MM
%S ANSI second as decimal number (00-61)
%t POSIX tab character
%T POSIX time in %H:%M:%S format
%U ANSI year week number (00-53) (Sunday start)
%w ANSI weekday as decimal number (0-6) Sunday=0
%W ANSI year week number (00-53) (Monday start)
%x ANSI locale-specific date
%X ANSI locale-specific time
%y ANSI year without century (00-99)
%Y ANSI year with century e.g. 1952
%z UNIX time difference to GMT +|-%H:%M
%Z ANSI timezone name, or null string if unknown
Defined in time.h

---

int strlen(const char *s)
Defined in string.h

---

char *strncat (char * to, const char * from, size_t length)
Defined in string.h

---

int strncmp(const char *string1, const char *string2,
size_t length)
Defined in string.h

---

char *strncpy(char *target, const char *source,
size_t length)
Defined in string.h

---

char *strpbrk(const char *s, const char *b)
Defined in string.h

---

char *strrchr(const char *s, int c)
Search a string backwards for a specified character
Defined in string.h

---

size_t strspn(const char *string, const char *set)
Get longest initial span of string that consists of characters defined in
set.
Defined in string.h

---

char *strstr (const char * string, const char *substring)
Locate first occurence of substring in string.
Defined in string.h

---

double strtod (const char *str, char **ptr)
Convert string to double.
Defined in stdlib.h

---

char *strtok(char *s, const char *b)
Search a string to find the first token character.
Defined in string.h

---

long strtol(const char *p, char **np, int base)
Convert string to long int

Convert string to long int.

Defined in stdlib.h

---

unsigned long strtoul (const char *str, char **ptr, int base)

Convert string to unsigned long int.

Defined in stdlib.h

---

size_t strxfrm (char *dest, const char *src, size_t length)

This is a dummy routine under C68 that performs just like the strcpy() routine. It is included for compatibility reasons.

Defined in string.h

---

int system (char * command)

Execute a command. The command string should consist of the name of the program to run followed by any parameters required separated from it by spaces. If multiple commands are supplied on the same line, then they should be separated by semicolons. Each command will be executed in turn until either one fails, or all the commands have been executed. If a command finishes with the & (ampersand) character then it will be run in parallel with the initiating program without waiting for it to complete.

The following built-in Unix style commands are also recognised:

cd	List current data directory setting.
cpd	List current program directory setting.
cd name	Set data directory to 'name'
cpd name	Set Program directory to 'name'
set	List settings of all environment variables.
set name	Displays value of environment variable 'name'
set name=val	Set environment variable 'name'

The value returned is the exit status of 'command'. If 'command' is NULL, then the value returned indicates whether a command processor is available (ie 1 in this case). If a command is being run in parallel, then the return value simply indicates whether the command was successfully started or not.

Defined in stdlib.h

---

time_t time(time_t * tptr)

Get the current system time. The time returned is a ANSI/Unix style time value rather than a QDOS one.

Defined in time.h

---

int tolower(int c)

Convert a character to lower case.

Defined in ctype.h

---

int toupper(int c)

Convert a character to upper case.

Defined in ctype.h

---

int ungetc(char c, FILE *fp)

Defined in stdio.h

---

int vfprintf (FILE *fp, char *format, char *params)

Perform formatted output to a specified file.

Defined in stdio.h

---

int vfscanf (FILE *fp, char *format, char *params)

Perform formatted input from a specified file.

Defined in stdio.h

---

int vprintf (char *format, char *params)

Perform formatted output to stdout.

Defined in stdio.h

---

int vscanf (char *format, char *params)

Perform formatted input from stdin.

Defined in stdio.h

Defined in stdio.h

---

int vsprintf (char *string, char *format, char *params)

Perform formatted output to a string.

Defined in stdio.h

---

int vsscanf (char *string, char *format, char *params)

Perform formatted input from a string.

Defined in stdio.h

---

size_t wcstombs (char *s, const wchar_t *pwcs, size_t n)

Convert a wide characater string to a multi-byte character string.

Defined in stdlib.h

---

int wctomb (char *s, wchar_t wchar)

Convert a wide character to a multi byte character.

Defined in stdlib.h

GLOBAL VARIABLES

---

extern int errno

Holds the error code if a routine reports a failure.

Defined in errno.h

AMENDMENT HISTORY

25 Aug 93	The itoa() description changed to come in line with accepted usage (parameters reversed).
30 Aug 93	Added new TIME routines difftime(), strftime() and tzset(). Added routines for converting between Wide Characters and Multi Byte Characters: mblen(), mbtowc(), mbstowcs(), wcstombs() and wctomb(). Added locale handling routines setlocale() and localeconv().
10 Oct 94	Reworked the documentation so that the ANSI specified routines are in their own file.

C68-specific Library Routines

INTRODUCTION

Use of the **libc68** library provides extensions are specific to the implementation of C68 on the QDOS or SMS operating systems. It will help you to exploit QDOS or SMS facilities to the full, but will mean that the programs you write will not be easy to transfer to other operating systems. You should bear this fact in mind when you decide to use the routines in the **libc68** library.

The implementation of C68 for QDOS and SMS also provides routines to allow the C programmer to access all the Operating System Call interfaces directly. These are documented in the LIBQDOS_DOC (using the QDOS names for such calls) or the LIBSMS_DOC (using the SMS names for the calls) files.

You do not have to make any special provison at the link stage if you want to include routines from the **libsms** library. The routines defined as being in this library are actually imbedded in the LIBC_A library which is automatically included at the end of the link by the LD linker. You must however include either

#include <qdos.h>

or

#include <sms.h>

in any program or module that use the routines defined in the **libc68** library. Which of the two you include is not material (you can include both!), and will probably be determined by whether you intend to use QDOS or SMS names for any calls you make directly to the operating system interface.

MIXING C and QDOS/SMS INPUT/OUTPUT

If you wish to be able to use both C and QDOS/SMS level input/output calls to refer to the same file/device then it is imperative that you issue a '**setbuf** ' call (defined in stdio_h) to disable internal buffering within the C standard input/output routines, or use the fflush() call before switching from C level I/O to QDOS/SMS level I/O. Failure to do this can result in input/output reacting in unexpected ways.

REFERENCE MATERIAL

The reference books listed below were used in preparing material for inclusion in this library:

"QL Technical Guide" by David Karlin and Tony Tebby

"QL Advanced User Guide" by Adrian Dickens

"QDOS Reference Manual" as published by Jochen Merz

LIBRARY ROUTINES

The following pages contain a list of all the routines contained in the C68 **libc68_a** library. These are routines that are specific to this QDOS or SMS implementations of C68. It is organised as a short list by function, and a longer list in alphabetical order.

FILE/DIRECTORY HANDLING

chddir chpdir fgetchid fnmatch

fqstat fusechid getcdd getchid

getcname getcpd getfnl opene

open_qdir qdir_delete qdir_read qdir_sort

qstat read_qdir usechid

SCREEN INPUT/OUTPUT

c_extop iop_outl

SOUND

beep do_sound

CONVERSION

cstr_to_ql d_to_qlfp i_to_qlfp l_to_qlfp

qlfp_to_d qlfp_to_f qlstr_to_c w_to_qlfp

STRING HANDLING

qstrcat qstrchr qstrcmp qstrcpy

qstricmp qstrlen qstrncat qstrncmp

qstrncpy qstrnicmp ut_cstr

MISCELLANEOUS

baud iscon isdevice isdirchid

isdirdev isnoclose keyrow poserr

qdos1 qdos2 qdos3 qinstrn

stackcheck stackreport waitfor _CacheFlush

_ProcessorType _super _superend _user

GLOBAL VECTORS

_bufsize _cmdchannels _cmdparams _cmdwildcard

_endmsg _memincr _memmax _memqdos

_mneed _oserr _pipesize _prog_name

_stack _stackmargin _sys_var

def_priority os_nerr os_errlist

---

void argfree (char ** argv[])

Routine to free all the memory that is associated with an argv[] style vector created using the argunpack() routine. This frees the memory associated with the argument strings as well as that associated with the argument vecotr itself.

---

char * argpack (char * argv[], int flag)

Routine to create a command line from an argv[] style vector. This is the complimentary routine to argunpack(). The command line will consist of the arguements from the argv[] vector separated by spaces. If the 'flag' parameter is set then it will be assumed that the command line is for a C68 program, and the arguments will be processed so that quotes are added around them if they contain white space, and any embedded non-printable characters are converted to C escape sequences. If the flag is not sent, then each argument is simply added unprocessed. The memory for the command line is allocated dynamically via malloc().

The value returned is the address of the resulting command line. If any error occurs (typically no memory left) then NULL is returned.

It is expected that the main use of this routine will be internally within other library routines, but it is made available for any system programmers.

---

int argunpack(char * cmdline, char ** argv[], int * argc,

int (* function)(char *, char *, int *))**

Routine to create an argv[] vector from a command line. This is the complimentary routine to argpack(). If any argument is surrounded by quotes

complimentary routine to argpack(). If any argument is surrounded by quotes these will be removed. Also, any enbedded C escape sequences will be converted into their internal values. The 'argc' parameter will be used to return a count of parameters put into the array less one (i.e. 0 means one value in the array).

The 'function' parameter is used to pass the address of a secondary routine that can be used to process further any argument before it is put into the array. A typical example of such a function might be the one that is used to do wild card expansion of parameters on the command line. If this function returns 0 then that means that it did nothing with the argument passed, and the argunpack() routine should add the value itself to the argv[] array. A return value of -1 inidicates an error occurred, and any positive value means that the function has handled the argument internally. The function' parameter can also be NULL to indicate that no additional processing needed of arguemnts.

The value returned is the number of arguments actually put into the array. If any error occurs (typically no memory left) then -1 is returned.

It is expected that the main use of this routine will be internally within other library routines, but it is made available for any system programmers. This is the routine that is used within the program startup code to parse the command line.

---

void beep(duration, pitch)

QDOS routine to make a quick beep, given duration in 50 (or 60) Hz ticks, and pitch (from 0 to 255).

---

int c_extop (chanid_t channel, timeout_t timeout,

int (*func), int number_of_params, ...)

This routine allows a routine to be called to do an extended operation on a QDOS or SMS channel. The parameters are passed in a way that is compatible with this routine being written in C (c.f. sd_extop()/iow_xtop() for assembler only routines).

The C routine will be called in supervisor mode, with the parameters specified by ... above passed to it on the stack. Each parameter is assumed to be no larger than 4 bytes in size (i.e. no structures are to be passed on the stack).

NOTE It appears that QDOS cannot correctly handle error codes being returned in D0. Therefore the only values that should be returned are 0 or -1 (for operation not completed). If it is desired to pass an error code back to the application program it must be done indirectly via one of the parameters.

---

int chddir(char *str)

Changes current destination directory (the one set by TK2 SPL_USE command in SuperBasic). If passed NULL then tries to go up a level. If passed a string starting with a device then replaces the current directory, else appends to current directory (adding _ at end if needed). Maximum length is 31 characters. Returns 0 if ok, !0 if failed.

---

int chpdir(char *str)

Changes current program directory (the one set by TK2 PROG_USE command in SuperBasic). If passed NULL then tries to go up a level. If passed a string starting with a device then replaces the current directory, else appends to current directory (adding _ at end if needed). Maximum length is 31 characters. Returns 0 if ok, !0 if failed.

QLSTR_t * cstr_to_ql(QLSTR_t * ql_string, char * c_string)

Routine to convert a C (zero terminated) string to a struct QLSTR (defined in qdos.h), a QL string with length first followed by the string. This routine is NOT safe to convert a C string in situ, eg. cstr_to_ql((QLSTR_t *)str, str) will fail badly (the C string will become corrupt). Returns the address of the QL string.

---

void do_sound(int duration, int pitch, int pitch2, int wrap,

int g_x, int g_y,int fuzz, int random)

QDOS call to make a sound. Parameters defined as for SuperBasic beep call.

QLFLOAT_t * d_to_qlfp(QLFLOAT_t * qlf, double val)

Routine to convert IEEE double precision (8 byte) floating point number to a QL floating point number. Returns the address of the QLFLOAT passed as the first parameter.

---

long fpatchid(FILE *fp)

long fgetchid(FILE *fp)

Returns QDOS channel id of FILE pointer. Returns -1L on error

Defined in stdio.h

---

int fnmatch(char *fname, char *wildname)

Non-recursive routine to match a QDOS wildcard. Similar to Unix style widlcard matching to make it more useful for GREP and 'C' programmers.

Examples of match

*_c matches names ENDING with _c only

(eg. test_c but NOT test_c_doc)

wom*_o matches wombat_o but NOT wombat_obj

*tes*_vi*_obj matches flp1_wombat_test_yy_vile_obj

but NOT flp1_wombat_testvile_obj

Returns 1 if match, 0 if no match

---

int fqstat(int fd, struct direct * stat)

QDOS specific variant of fstat() call. Normally it would be recommended that you used the fstat() call instead as this is more portable. Gets the file information from QDOS, given a level 1 file descriptor. Exactly same information as in a QDOS directory entry (Note times are in QDOS format, not C format). The structure 'direct' is defined in 'qdos.h'.

Returns values:

0 success

-1 Standard C error code set in errno (as defined in errno.h)

other QDOS error code (as defined in qdos.h).

FILE * fusechid (chanid_t channel)

Create a Level 2 File Pointer for a file opened at Level 0 (the QDOS level) via the io_open() call. Also creates a level 1 file descriptor entry. Must NOT be called more than once for a given file.

Return values:

+ve FILE pointer

NULL failed - details in errno

---

char *getcdd(char *str, int size)

Gets current destination directory path (as set by TK2 SPL_USE command) into buffer str. If str == NULL then allocates a buffer of length size using malloc and returns address of it. Returns NULL on error, else address where name is stored.

---

chanid_t getchid(int fd)

Gets QDOS channel id for level 1 file descriptor.

Return values:

-1 error occurred - details in errno

+ve QDOS channel id

---

char *getcname (chanid_t channel, char *buffer)

Obtains the name of a device associated with a QDOS channel and places it in the buffer.

Return values:

+ve Pointer to the name

NULL error occurred - details in errno.

---

char *getcpd(char *str, int size)

Gets current program directory path (as set by TK2 PROG_USE command) into buffer str. If str == NULL then allocates a buffer of length size using malloc and returns address of it.

Return values:

NULL error occured - details in errno.

+ve address where name is stored.

---

int getfnl(wcard, fna, fnasize, attr)

char *wcard; /* Wild card string to use, or NULL for all the files in the data directory */

char *fna; /* Area to hold returned list of file names */

unsigned int fnasize; /* Size of file name area */

int attr; /* Search attributes. Can be added

together to provide criteria.

0 - all files (QDR_ALL)

1 - data only (QDR_DATA)

2 - prog only (QDR_PROG)

4 - directory only (QDR_DIR)

Symbolic names defined in qdos.h */

Lattice compatible routine to get a list of filenames, separated by '\0' character. List terminated by an additional '\0' character.

Return values:

-1 error occurred

other number of names read.

Defined in stdlib.h. See also read_qdir().

---

int iop_outl (chanid_t channel, timeout_t timeout,

short, short, short, void *)

This is the call that sets the outline window for a Pointer Environment. It is included in this library as it is the one call that need to be issued to make a program that is not otherwise aware of the pointer environment function correctly in that environment.

For more details refer to the LIBQPTR_DOC file provided as part of the QPTR library.

Note that the default console initialisation routines supplied with C68 will automatically issue a call to set the window outline to the size as defined in the '_condetails' global variable (see end of this document).

---

int iscon(chanid_t long channel, timeout_t timeout)

returns 1 is is con, 0 if not

---

int isdevice(char *str, int *extra)

Routine to check if a string starts with a device name. TRUE if it is, with extra info in the 'extra' parameter passed as well as the name, 0 if it's not. Actually searches system lists. *extra can be DIRDEV (device is on directory driver lists) or DIRDEV | NETDEV (device is on a network - may not be directory device on remote machine).

DIRDEV and NETDEV are defined in qdos.h.

QDDEV_LINK_t * isdirchid (chanid_t channel_id)

Routine to find out if a channel belongs to a directory device or not. If not, NULL is returned. If it does, then a pointer to the Device Driver Definition block is returned. This can then subsequently be used to find out the device type if required by looking at the name field in this Device Driver Definition block.

---

int isdirdev (char *str)

Routine to check if a string starts with a name corresponding to a directory device. Returns 0 if not. The value returned has the same meaing as the 'extra' parameter returned by the isdevice() routine.

---

int isnoclose (int file_descriptor)

Used to determine if the channel associated with a level 1 file descriptor was passed to this job on the stack (via the command line). Return values are:

-1 file does not exist

1 channel for this file was passed on the stack

0 channel for this file was not passed on the stack

QLFLOAT_t * i_to_qlfp(QLFLOAT_t * qlf, int i)

Fast routine (faster than inbuilt QDOS routine) to convert a integer into a QL floating point number. Returns the address of the QLFLOAT passed as the first parameter.

---

int keyrow (int row)

QDOS routine to read the QL keyboard directly. Equivalent to SuperBasic keyrow with all attendant warnings. Does not set _oserr.

QLFLOAT_t * l_to_qlfp(QLFLOAT_t * qlf, int i)

Fast routine (faster than inbuilt QDOS routine) to convert a long integer into a QL floating point number. Returns the address of the QLFLOAT passed as the first parameter.

---

int opene(char *name, int mode, int paths)

Routine to search more than just the default directory if name does not start with a device. If it does then that is opened, else if :-

paths == 3 search program directory, then data directory

== 2 just search program directory

== 1 search data directory first, then program directory

== 0 just search data directory (as open())

Returns -1 on error, valid fd if OK.

Defined in fcntl.h

---

chanid_t open_qdir(char *name)

Opens a directory on a device. Returns a negative value (the QDOS error code) if an error occurred at the QDOS level, 0 if any other error occurred (in which case 'errno' contains the error code) or a positive channel id on success.

---

int poserr(char *s)

The QDOS specific equivalent of the standard C 'perror' routine. Prints the error text relating to the QDOS error code in _oserr.

---

void qdir_delete(DIR_LIST_t * list)

Deletes all space allocated by a call to the qdir_read() routine.

DIR_LIST_t * qdir_read(devwc, stext, attr)

char *devwc; /* Device and wildcard */

char *stext; /* Sort text */

int attr; /* File types to get

0 = all,

1 = data,

2 = prog,

4 = directory */

Routine to open, read and sort a QDOS directory. Sort text is same as QRAM, N(ame) U(se) S(ize) D(ate) T(ime) lower case reverses sense of search. The DIR_LIST_t structure is defined in sys_qlib.h. All space for directory entries and names is allocated via malloc() - it should be released when you have finished with it by calling qdir_delete().

Return values:

NULL No match found (or error occured if errno set)

other Pointer to list

DIR_LIST_t * qdir_sort(DIR_LIST_t *list,

char *stext, char (*dcomp)())

DIR_LIST_t * list; /* Existing linked list */

char *stext; /* Sort parameters */

char (*dcomp)(); /* Compare routine (default routine

used if dcomp == NULL)*/

Routine to sort linked list of extended QDOS directory structure. Returns pointer to first of list.

Sort text is string containing:

N or **n** sort on ascii name.

U or **u** sort on file usage.

S or **s** sort on file size.

D or **d** sort on file date.

T or **t** sort on file time.

Uppercase = ascending sort, Lowercase = descending

If strlen(stext) > 1 then each sort is done in turn.

A default compare routine is used internally by qdir_read(). This is described below in case anyone wants to write better compare routines.

Specification of dcomp is:

int (*dcomp)(DIR_LIST_t * d1, DIR_LIST_t *d2,

char *sort_text)

Return value indicates comparison result:

+ve d1 > d2

0 d1 == d2

-ve d1 < d2

---

long qdos1(REGS_t *in, REGS_t *out)

long qdos2(REGS_t *in, REGS_t *out)

long qdos3(REGS_t *in, REGS_t *out)

Lattice compatible routines to call specific operating systemp traps with registers set up as in a REGS_t structure. Not normally needed with C68 as there are routines in the libraries to call most QDOS and/or SMS traps directly. These routines cater for any that might be missing, and also provide compatibility with QLC which used these routines for accessing QDOS facilities. Returns the value of register D0.

---

int qfork(...)

Starts another process concurrently with the calling one. The new process start with a default priority found in external variable **_def_priority** . Returns process id of new process or error code. Sets errno (and if relevant _oserr). The arguments (other than 'owner') have the same meaning as in the **exec()** and **fork()** family of calls.

These are variants of the **fork()** family of calls (that are defined in LIBUNIX_DOC). The difference is that the **qfork()** variants allow the owner of the new process to be specified whereas with the **fork()** set the current job is always the owner. This is important under QDOS or SMS if you do not want the daughter job to be automatically terminated by the operating system when the parent job terminates. If you specify zero as the parent job then the daughter job is in complete control of its own destiny!

pid_t qforkv(jobid_t owner, char * name, int * file_desc,

char * argv[])

pid_t qforkvp(jobid_t owner, char * name, int * file_desc,

char * argv[])

pid_t qforkl(jobid_t owner, char * name, int * file_desc,

char * argvs, ...)

pid_t qforklp(jobid_t owner, char * name,int * file_desc,

char * argvs, ...)

The directories searched in each case are as follow:

qforkv program directory only

qforkvp program directory and then data directory

qforkl program directory only

qforklp program directory and then data directory

The **qforkl()** and **qforklp()** routines must have a NULL parameter to terminate their parameter lists.

---

int qinstrn (char * string, int max)

Function to type a C style string into the current keyboard queue (c.f. the Turbo Toolkit command TYPE_IN). On success returns the number of characters typed in, on failure returns a QDOS error code.

---

double qlfp_to_d (QLFLOAT_t * qlfp)

Routine to convert the 6 byte representation of floating point numbers used on QDOS and SMS systems to the IEEE 8 byte floating point format used internally by C68 for doubles.

---

long qlfp_to_f (QLFLOAT_t * qlfp)

Routine to convert the 6 byte repreentation of floating point numbers into the bit pattern corresponding to an IEEE floating point number as a long. This is NOT the routine to use if you merely wish the result to be assigned to a C 'float' variable - use qlfp_to_d() instead.

---

char *qlstr_to_c(char *c_string, QLSTR_t * ql_string)

Routine to convert a QDOS or SMS string (length first, followed by string) (the QLSTR_t strucutre is defined in sys/qlib.h which is included by both qdos.h and sms.h) to a C string (zero terminated). Note that this routine is safe to call to convert a QDOS or SMS string in situ eg.

qlstr_to_c(q_string, (char *)q_string)

as nothing is corrupted.

---

int qopen (const char *, int mode, ...)

This is a variant of the **open()** routine that is specifically designed to make it easy to handle filenames that originate on foreign systems. These foreign systems often have special characters in their filenames to indicate sub-directories or file extensions. On a QDOS or SMS system one would typically use underscores for both these purposes. This routine handles an automatic between these different types of name in a relatively transparent manner.

If the filename supplied does not contain any of the special characters ' . ' (fullstop), ' **/** ' (forward slash) or ' **** ' (backward slash) then this routine is functionally identical to the open() routine. If the filename supplied does contain any of the special characters then the way it operates depends on whether the file is being opened with a READ ONLY mode or some variant of a WRITE mode:

READ A copy of the filename is made with the special characters replaced by underscores and an attempt made to open the file with this revised filename (i.e. the typical QDOS/SMS variant is tried first). If this fails, then the original name as supplied is tried as well.

WRITE A check is made to see if a file with the name as supplied is present, and if so this name is used (i.e. the foreign name is tried first). If such a file is not present then a copy of the name is made and the special characters replaced by underscores. This name is then used to open the file.

Note.

The **qopen()** routine would typically be used in conjunction with the **_Open** vector described later in this document.

---

int qstat (char *name, struct qdirect *buffer)

Routine to get file information given the filename. This is a QDOS and SMS specific variant of the stat() call. It is recommended that you try and use the stat() call in preference as this is more portable. The qdirect structure is defined in sys/qdos.h which is included by both qdos.h and sms.h.

Return values:

0 Success

-1 Standard C error code set in errno (as defined in errno.h).

other QDOS error code (as defined in qdos.h)

QLSTR_t * qstrcat (QLSTR_t * target, const QLSTR_t * source)

Concatenate two QDOS or SMS strings. Similar to the C routine strcat() except that it operates on QDOS and SMS strings. The QLSTR_t structure is defined in sys/qlib.h which is included by both qdos.h and sms.h. The target string will also have a NULL byte appended to the end (although this will not be included in the length count) so that it is possible to treat the text part as a C string.

---

int qstrchr (const QLSTR_t * target, int ch)

Search a QDOS or SMS string for a specified character. Similar to the C routine strchr() except that it operates on QDOS and SMS strings. The QLSTR_t structure is defined in sys/qlib.h which is included by both qdos.h and sms.h. The value returned will be the address of the character, or NULL if the character was not found.

---

int qstrcmp (const QLSTR_t * string1,const QLSTR_t * string2)

Compare two QDOS or SMS strings for equality. Similar to the C routine strcmp() except that it operates on QDOS and SMS strings. The QLSTR_t structure is defined in sys/qlib.h which is included by both qdos.h and sms.h. The QDOS/SMS collating order is used to determine the less than/greater than return conditions.

QLSTR_t * qstrcpy (QLSTR_t * target, const QLSTR_t * source)

Copy a QDOS or SMS string. Similar to the C routine strcpy() except that it operates on QDOS and SMS strings. The QLSTR_t structure is defined in sys/qlib.h which is included by both qdos.h and sms.h. An additional NULL byte is added to the end of the target string (although not included in the length count) so that it is possible to treat the text part as a C string. This extra byte must be allowed for in determining the required size of the target area.

---

int qstricmp (const QLSTR_t * string1,

const QLSTR_t * string2)

Compare two QDOS or SMS strings for equality ignoring case. Similar to the

C routine stricmp() except that it operates on QDOS and SMS strings. The QLSTR_t structure is defined in sys/qlib.h which is included automatically by qdos.h or sms.h. The QDOS/SMS collating order is used to determine the less than/greater than return conditions.

---

int qstrlen (const QLSTR_t * target)

Get the length of a QDOS or SMS string. Similar to the C routine strlen() except that it operates on QDOS and SMS strings. The QLSTR_t structure is defined in sys/qlib.h which is included autoamtically by qdos.h and sms.h.

QLSTR_t * qstrncat (QLSTR_t * target, const QLSTR_t * source,
size_t maxlength)

Concatenate one QDOS or SMS string to another one up to a specified length. Similar to the C routine strncat() except that it operates on QDOS and SMS strings. The QLSTR_t structure is defined in sys/qlib.h which is included autoamtically by qdos.h and sms.h. The target string will also have a NULL byte appended to the end (although this will not be included in the length count) so that it is possible to treat the text part as a C string.

---

int qstrncmp (const QLSTR_t * string1,
const QLSTR_t * string2, size_t maxlength)

Compare two QDOS or SMS strings for equality up to a maximum length. Similar to the C routine stricmp() except that it operates on QDOS and SMS strings. The QLSTR_t structure is defined in sys/qlib.h which is included automatically by qdos.h and sms.h. The QDOS/SMS collating order is used to determine the less than/greater than return conditions.

QLSTR_t * qstrncpy (QLSTR_t * target,
const QLSTR_t * source, size_t maxlength)

Copy a QDOS or SMS string up to a maximum length. Similar to the C routine strncpy() except that it operates on QDOS and SMS strings. The QLSTR_t structure is defined in sys/qlib.h which is included automatically by qdos.h and sms.h. An additional NULL byte is added to the end of the target string (although not included in the length count) so that it is possible to treat the text part as a C string. This extra byte must be allowed for in determining the required size of the target area.

---

int qstrnicmp (QLSTR_t * string1, QLSTR_t * string2,
size_t maxlength)

Compare two QDOS or SMS strings for equality ignoring case up to a specified length. Similar to the C routine strnicmp() except that it operates on QDOS and SMS strings. The QLSTR_t structure is defined in sys/qlib.h which is included automatically by qdos.h ad sms.h. The QDOS/SMS collating order is used to determine the less than/greater than return conditions.

---

int read_qdir(chid, devwc, ret_name, ret_dir, attr)

chanid_t chid; /* QDOS channel id for directory */

char *devwc; /* device and wildcard */

char *ret_name; /* Name to return */

struct direct *ret_dir; /* Directory structure to read into */

int attr; /* Types to read:

0 = all,

1 = data,

2 = prog,

4 = directory */

Reads the next directory entry matching a specified wildcard

and attribute. If first part of wild matches _dir_ then only last part of name returned.

Return values:

1 Success

0 End-of-file reached

-1 Error as indicated by errno

See also getfnl().

---

int sendsig(chid,jobid,signo,priority,uval)

Low level routine to send a signal to the SIGNAL device driver. Returns 0 on success, or QDOS/SMS error code on failure.

Defined in signal.h

---

int set_timer_event(struct TMR_MSG *msg)

Signal related routine that returns msg.len if a previous event was cancelled, 0, or QDOS error

Defined in signal.h

---

int sigcleanup()

Routine that should be called only when leaving a signal

handler through longjmp(). It inhibits reasigning of handler and sigprocmask and calls _CheckSig(). It is normally better to call the Posix defined routines sigsetjmp() and siglongjmp() instead of setjmp() and longjmp() as then this routine is not required. Returns 0 on success, QDOS/SMS error code on failure.

Defined in sys/signal.h

---

int stackcheck ()

This routine acts like **stackreport() ,** except that if the margin is breached, a 0 value is always returned (rather than a negative value). This means you can easily test for failure using **assert** statements of the form

assert(stackcheck());

in your program, and an assert error message is generated if it fails.

---

long stackreport()

Report the current amount of stack available before the safety margin (as specified by the global variable **_stackmargin**) is reached. A negative value means that you are below the safety margin by the specified amount, and are could well be corrupting your data areas. A program crash (or even system crash if you are unlucky) is probably imminent!

---

int strfnd(char *tofind, char *tosearch)

Find the position of string 'tofind' in string 'tosearch' doing case independent match. Returns -1 if not found, position in string if found.

Note that if you want a case dependant version you should use the Unix compatible strfind() routine (defined in LIBUNIX_DOC).

Defined in string.h

---

void strmfe (char * newname,

const char * oldname, const char * extension)

Lattice compatible routine to take a filename, remove any existing extension, and then to add the given extension.

Defined in string.h

---

void strmfn (char * newname, const char * drive,

const char * path, const char * basename,

const char * extension)

Lattice compatible routine to build up a filename from its components. Any required undersocre seperator characters will be added automatically. Any of the components can be a zero length string if not required.

Defined in string.

---

void strmfp (char * newname, char * path, char * name)

Lattice compatible routine to build a filename from its path and base name. If needed an undersocre character will be added between the 'path' and 'name' components. The 'path' string can be an zero length string.

Defined in string.h

---

int usechid (chanid_t channel)

Create a Level 1 file descriptor for a file opened at level 0 (ie the QDOS level using io_open()). Must not be called more than once for a given file.

Returns file descriptor if successful, -1 on failure.

QLFLOAT_t * w_to_qlfp (QLFLOAT_t * qlf, int w)

Routine to convert a short integer (word) to a QL floating point number. This routine is included mainly for completeness as normally you would use the i_to_qlfp() routine.

---

int waitfor (jobid_t jobid, int * ret_value)

Wait for the specified job to terminate. If the 'ret_value' parameter is not NULL, then it should point to the address at which the exit code of the specified job should be put. Returns 0 on success, -1 if specified job could not be found.

---

void _CacheFlush (void)

Routine to force a flush of the cache on 68030 (or higher) processors - will do nothing on 68020 or less. Needs to be used if you ever have self-modifying code. Ths means that it should very rarely be used in practise!

---

int _ProcessorType (void)

Routine to determine the type of processor you are running on. If the system variable that specifies the processor type is set, then this value is returned. If the system variable is not set then tests are done to determine the processor type, the value is stored in the system variable and also returned. The values returned will indicate the basic processor type as follws:

$00 68000/68008

$20 68020

$30 68030

$40 68040

In addition the following bits can be 'or'ed to the above values to indicate special features:

$01 Internal MMU

$02 68851 MMU

$04 Internal FPU

$08 68881/68882 FPU

Experience has shown however, that one cannot guarantee that the bits inidcating extra features will always be set up - and there is quite a bit of code around that works on the assumption they will not be set up.

---

void _super()

Routine to go into supervisor mode. You **MUST** return to user mode before exiting the function in which you went into supervisor mode or you will probably crash the machine.

WARNING This function should be used with great care, and only if absolutely essential.

---

void _superend()

Routine to go back into user mode after having been in Supervisor mode. Does not check if any signals have occurred. Complementary function to **_super()** (see also _user() routine).

---

void _user()

Routine to go back into user mode after having been in Supervisor mode. Checks if any signals have occurred while in supervisor mode, and if so handles them. Complementary function to **_super()** (see also __superened() routine).

STRUCTURES, MACROS and TYPEDEFs

These are various definitions in the sys/qlib.h header file that are used when refering to QDOS or SMS based systems. This header file is included automatically by the qdos.h, sms.h and qptr.h header files.

JOBHEADER and **JOBHEADER_t**

These are the structure name and typedef respectively that are used to refer to a QDOS/SMS job header.

QFLOAT and **QFLOAT_t**

These are the structure name and typedef respectively that are used when refering to QL/SMS format floating point numbers.

QLRECT and **QLRECT_t**

These are the structure name and typedef respectively that are used to define the width, height and origin of a rectangular area on a screen.

QLSTR and **QLSTR_t**

These are the structure definition and typedef respectiviely used to refer to QL/SMS string types. They are defined in the sys/qlib.h header file.

QLSTR_DEF (name, length)

Macro to define the space for a QL/SMS style string in a QLSTR style structure. Typically used in a statement of the form:

QLSTR_DEF (string_name,20);

You can also give the string an initial value by using a statement of the form:

QLSTR_DEF (string_name,20) = {5,"Hello");

However if the string will never be changed, you will find it easier to use the QLSTR_INIT macro.

QLSTR_INIT (name, "value")

Macro to define a constant initialised QL/SMS style string. Typically used in a statement of the form:

QLSTR_INIT (string_name,"Hello");

The space allocated will allow for the NULL byte that is used to terminate a C style string, but the NULL byte will not be included in the count of characters in the QL/SMS part.

If you use this macro inside a function then you need to precede it with the **static** keyword.

TIME_QL_UNIX (ql_time_in_seconds)

Macro to convert a QDOS/SMS time in seconds (measured since 1st January 1961) into a Unix time in seconds (measured since 1st January 1970).

TIME_UNIX_QL (unix_time_in_seconds)

Macro to convert a Unix time in seconds (measured since 1st January 1970) into a QDOS/SMS time in seconds (measured since 1st January 1961).

WINDOWDEF and **WINDOWDEF_t**

These are the structure and typedef respectively that are used to define the details of a screen window.

GLOBAL VARIABLES

The following are glabal variables that are available to user programs. Some of them are for information only while others can be set in user programs to control certain default settings of elements of the C68 run-time environment. In these cases, if the user does not provide a value, then the specified default values will be used.

---

extern long _def_priority

Used to set priority of new jobs. Default is a value of 32.

---

extern int os_nerr

Number of QDOS error messages catered for in 'os_errlist' table.

---

extern char *os_errlist[]

Table giving text for all the standard QDOS error codes. Use the negation of the QDOS error code (to convert it to a positive number) as an index into this table to get the text for a particular error code.

---

extern WINDOWDEF_t _condetails

This contains the definition details for the initial console window. The WINDOWDEF_t type refers to a structure that is defined in sys/qlib.h The default values are equivalent to a C statement of the form:

```
WINDOWDEF_t _condetails =
{
2, /* border colour (red) */
1, /* border width */
0, /* paper (black) */
7, /* ink (white) */
464. /* width (pixel) */
180, /* height (pixels) */
24, /* x origin */
26 /* y origin */
};
```

This global variable is used by the consetup_default() and consetup_title() routines to determine the console details.

---

extern char _copyright[]

This variable is used by the consetup_title() routine. It inserts this string at the left side of the menu bar. The default value is a zero length string, but the user can define his own text.

---

extern char * _endmsg

The message that will be used when a program closes down. Default is "Press a key to exit". After displaying the message, the program waits for a keypress. Setting this pointer to NULL will mean that the program exits without displaying any message.

---

extern timeout_t _endtimeout

The timeout that will be used when displaying the closedown message and waiting for a response. The default is -1 which means wait forever. Positive values are the number of 1/50 second units to wait.

---

extern long _memincr

Sets the minimum increment in which new memory allocations will be made from the stack. Default value is 4K bytes.

---

extern long _memmax

Sets the maximum memory that a program is allowed to allocate. Default is as much as the program wants.

---

extern long _memfree

Sets the amount of memory that must always be left for QDOS or SMS when trying to allocated additional memory for a program. Default is 20K bytes.

---

extern long _mneed

Sets program initial memory allocation. A negative value can be set which means allocate all the memory **except** this amount. Default is 8K bytes.

---

extern long _oserr

Used to return QDOS/SMS error codes for some of the QDOS/SMS trap and/or vector calls. It can also be set when an error return is made from a standard C level routine with the **errno** global variable set to the value **EOSERR** (as defined in errno.h).

---

extern long _pipesize

Sets default pipe size.

---

extern char _prog_name[]

Sets default program name. Default is a name of C-PROG.

---

extern char _Qopen_in[]

This is the list of special characters that are checked for by the **qopen()** library routine. It should be NULL terminated. Its default value is the string **"/.\\"** .

---

extern char _Qopen_out[]

This is the list of what each character that is found in the **_Qopen_in** string should be converted to. It must be at least the same length as **_Qopen_in** or the effect is undefined. Its default value is the string **"___"** .

---

extern long _stack

Sets program stack size. Default is 2Kb bytes.

---

extern long _stackmargin

Sets the default value for the 'stackcheck' routine to start reporting failures. Default is 256 bytes.

---

extern char *_sys_var

Base of system variables. Set when the program starts-up.

---

extern char _version[]

This is a string used by the consetup_title() routine. It is inserted at the right hand end of the menu bar. Default value if this string is not defined explicitly in the users program is a zero length string.

GLOBAL VECTORS

The following are glabal vectors that can be set in user programs to control certain default actions of the C68 run-time environment. If the user does not provide a value, then the specified default values will be used.

N.B. Setting other values that are specified here can have undefined effects and are very likely to cause a system crash.

---

extern long (*_cmdchannels)()

This can be set to NULL if the program cannot be passed channels directly from SuperBasic. Default is to include code to allow channels to be accepted from SuperBasic.

---

extern void (*_cmdparams)()

This can be set to NULL if the program does not take any parameters. This will stop code for parsing the command line being included. Default is to include code for parsing the command line.

---

extern void (*_cmdwildcard)()

This can be set to specify the routine to expand wild cards if they are found in the command line. Default is NULL which means that wildcards are not expanded. The routine **cmdexpand()** is provided which will simulate the filename expansion that is done by the Unix shell.

---

extern void (*_consetup)()

This contains a pointer to the routine that will be called to initialise the console window on program startup. It will only be called if the console channel was NOT passed on the stack from another program.

The default routine **consetup_default()** merely clears the window and puts a border around it. The routine **consetup_title()** is also provided in the standard library. This will additionally provide a title bar at the top of the window (c.f. the **_copyright** and **_version** global variables).

If this vector is set to NULL, then no default initialisation is done.

Alternatively, the user can provide his own alternative routine. See QDOSC68_DOC for more details.

---

extern long (*_conread)(UFB_t * uptr,

void * buffer, long length)

This is a pointer to a routine that will handle any input translation for console/screen devices of any special characters during a read. The supplied default routine acts on the following special characters:

CTRL-D Treated as EOF

CTRL-X Treated as "Kill Job"

This vector can be set to NULL if console input translation is definitely not required. This will cause the relevant code to be omitted from the program.

This vector can be set to point to an alternative routine if more comprehensive input translation is required. The value returned is the number of characters read into the buffer.

---

extern long (*_conwrite)(UFB_t * uptr,

void * buffer, long length)

This is a pointer to a routine that will handle any output translation for console/screen devices of any special characters during a write. The supplied default routine handles the ANSI C specified escape sequences.

This vector can be set to NULL if console output translation is definitely not required. This will cause the relevant code to be omitted from the program.

Alternatively, if more sophisticated output translation is required then a user routine can be substituted. The return values from this routine are treated as follows:

0 an error occurred

+ve output the specified number of characters from the buffer without translation

translation.

-ve the specified number of characters from the buffer required special translation which has been done.

---

extern int (*_Open)(const char * name, int mode, ...)

This is a pointer to the routine that will be used for any **open()**, **fopen()** or **stat()** routines in the program. By default this points to a standard internal library routine that implements the **open()** call. If the special additional actions carried out by the **qopen()** routine are required then this can be invoked by setting the **_Open** vector as follows:

```
#include <fcntl.h>

int (*_Open)(const char *, int, ...) = qopen;
```

If you wish to write some other variant of the **open()** call, then look at the source of the **qopen()** module for an example of how to go about this. --

---

extern int (*_readkbd) (chanid_t channel, timeout_t timeout,

char *, byte_read);

This is a pointer to the routine that is used to read the keyboard. Normally it would point to the standard operating system call for reading a byte.

Setting this to another value allows you to write a routine that can intercept keyboard input before it is passed back to the main C program. For an example of such a routine and how it might be used see the **readmove()** routine provided in the QPTR part of the standard C library.

CHANGE HISTORY

16 Jun 93	Added descriptions for the new string handling routines qstrcat(), qstrchr(), qstrcmp(), qstrcpy(), qstrlen(), qstricmp(), qstrncat(), qstrncmp(), qstrncpy(), qstrnicmp(), ut_cstr()
10 Jul 93	Description of calls amended to remove the statement that they set the _oserr global variable (where this is no longer true).
31 Dec 93	Documented the _copyright' and '_version' global variables.
24 Jan 94	Removed all references to the direct QDOS and SMS operating system calls. These are now documented in the LIBQDOS_DOC and LIBSMS_DOC files.
03 Sep 94	Added descriptions of the argfree(), argpack() and argunpack() routines.
20 Jan 95	Added descriptions of the qopen() routine and the associated '_Open' vector.
10 Feb 95	Documented the qfork() family of routines.
16 Apr 95	Added descriptions of the more important structures and typedefs that are defined in the sys/qlib.h header file.
28 Sep 95	Added description of _endtimeout global variable. Updated to reflect implementation of Richard Zidlicky's signal handling extension.
07 Dec 96	Added description of strfnd() routine, amended to be always case independent.
16 May 98	Added descriptions for the _CacheFlush() and _ProcessorType() routines.

C68 Curses library

INTRODUCTION

This is the C68 Curses library. Many thanks must go to Keith Walker for the work that he has done in producing this library.

NOTE. If anyone produces more complete documentation for some (or all) of these routines, then please feed them back for inclusion into later releases of C68. Also please feed back any errors that you notice in the documentation.

The purpose of the cursor library is to get a portable way of writing programs that update character screens. It allows the programmer to manipulate such screens without the need to know the details of the control sequences recognised by any particular screen type.

ENVIRONMENT VARIABLES

Programs that are going use the Curses library require the following two environment variables to be set:

```
TERM="qdos"

TERMINFO="flp1_terminfo"
```

The first gives the terminal "type", and the second one the place where the associated "terminfo" file can be found. The disk supplied includes a "terminfo" file for a terminal by the name of "QDOS".

"terminfo" file for a terminal by the name of "QDOS".

Also, the start-up code that is automatically linked in will set the two environment variables LINES and COLUMNS to amtch the size of the window allocated (default size is 80 columns by 24 lines).

LIBRARY ROUTINES

The following is an alphabetical list of all the routines contained in the C68 libcurses_a Curses library. No detail is given on any of the routines except those specific to the QDOS implementation. It is assumed taht you will have more detailed documentation on curses if you are trying to write your own program using this library.

Curses Routine Name	Curses Manual Page Name
addch	curs_addch
addchnstr	curs_addchnstr
addchstr	curs_addchstr
addnstr	curs_addstr
addstr	curs_addstr
attroff	curs_attr
attron	curs_attr
attrset	curs_attr
baudrate	curs_termattrs
beep	curs_beep
bkgd	curs_bkgd
bkgdset	curs_bkgd
border	curs_border
box	curs_border
can_change_color	curs_color
cbreak	curs_inopts
clear	curs_clear
clearok	curs_outopts
clrtobot	curs_clear
clrtoeol	curs_clear
color_content	curs_color
copywin	curs_overlay
curs_set	curs_kernel
def_prog_mode	curs_kernel
def_shell_mode	curs_kernel
del_curterm	curs_terminfo
delay_output	curs_util
delch	curs_delch
deleteln	curs_deleteln
delscreen	curs_initscr
delwin	curs_window
derwin	curs_window
doupdate	curs_refresh
dupwin	curs_window
echo	curs_inopts
echochar	curs_addch
endwin	curs_initscr

N.B. THIS LIST IS NOT YET COMPLETE

QDOS SUPPORT FUNCTIONS

The following routines are present in the C68 LIBCURSES_A library to handle the interface between the Curses routines and the standard C library routines.

They each replace routines of the same name in the standard C68 LIBC_A library. These replacment versions have higher and/or slightly different functionality to those in the LIBC_A library (and are therefore bigger).

---

struct WINDOWDEF _condetails

This routine describes the details of the window to be used as the basic console window. The default values are:

x_origin		
y_origin		
window height	244	(24 lines)
window width	484	(80 characters)
border width	1	
border colour	2	(red)
paper	0	(black)

ink 7 (white)

This give (in high resolution mode) a window with a useable area of 80 characters by 24 lines. This is the size that many curses based programs assume by default. Those that do not will almost certainly use the LINES and COLUMNS environment variables.

The WINDOWDEF structure is defined in qdos.h

---

struct termios _condevice

This data area is used by the other QDOS specific routines in the Curses library to hold information relating to the state of the Console device.

---

int _conread (struct UFB * uptr, char * buf, int length)

This routine is used to handle reading from the console. It recognises and actions the various input modes that curses tries to use.

---

int _conwrite (struct UFB * uptr, char * buf, int length)

This routine is used to handle writing to the console. It recognises and actions the various control sequences that curses tries to use. For more details refer to the **terminfo** file provided with this library.

---

void _initcon (void)

This routine completes the initialisation of the console device. In particular it sets the LINES and COLUMNS environment variables to reflect the actual size of the screen allocated.

Source Code Debugging Library

NAME

debug - source code debugging library.

COPYRIGHT

SYNOPSIS

```
#include <debug.h>
void dbg_init ();
void dbg (label, detail, trace_format, ...);
void dbg_print (.....);
void snap1 ();
void snap2 ();
void snap3 ();
char * label;
int detail;
char * trace_format;
```

DESCRIPTION

One of the problems areas when developing C programs is the process of debugging them. This can be extremely difficult and time consuming. The debugging system described here simplifies this task by providing a mechanism for monitoring the execution of a C program as it is running.

This debugging system is unusual in that it is implemented completely at the source code level, and does not need any low level machine code debugging support to make it work. This makes it relatively easy to port this system to other operating systems.

Features that are supported include:

- Dynamic control of stop points
- Dynamic control of trace frequency and detail
- Watch points
- Stack checking options
- Support for user defined snapshots
- Memory examination facilities
- Logging of a complete session
- Support for the MALLOC debugging library.

The debugging system (known as **dbg**) is invoked by including special format function calls in the program as you develop it (if you adopt the technique described below, it is not necessary to remove such calls when you want to

produce a version of the program with no imbedded debugging code). You will also need to link in the library **libdebug_a** which includes the code to support the debugging system. You can do this by specifying the parameter **-ldebug** to the linker. If you intend to use the malloc debugging library **libmalloc_a** in conjunction with **libdebug_a** then the order of linking is important. The parameters to the linker MUST be in the order

-lmalloc -ldebug

or you will get multiply defined symbols occuring.

At the start of your program, you should initialise the debugging system by a function call of the format

(void) dbg_init();

This will initialise the internal data structures of the **dbg** system, and pass control to the user. If you omit this call, then the **dbg** system will generate an automatic call to this routine when it is first entered. This may, however, mean that not all levels of stack can be traced and checked.

At appropriate points throughout the program you can then include calls of the form

dbg (label,detail_index,trace_format,...)

where **label** is a string identifying this particular call to the debugging system., **detail_mask** is an integer indicating the mask that must be satisfied to activate this call to **dbg** (this number must always be greater than zero) and **trace_format** is a 'printf' type format specification string. It may be a NULL pointer if no trace information is to be generated from this call to **dbg** . If a non-null pointer is provided, the string it identifies will be used to control the printing of the arguments following it. ,... is an optional, variable length list of variable identifiers as in 'printf' calls. These are variables that will be traced if this particulr call to **dbg** is activated as a trace point.

There may well be situations in which the trace facilities offered by **dbg** are either not suitable or are not comprehensive enough. A facility is available for the user to provide a number of 'snapshot' routines. These can then be invoked from **dbg** by use of the SNAP command as described later.

These snapshot routines must be of the form

```
void dbg_snap? ()
{
  user supplied code
}
```

where 'dbg_snap?' is 'dbg_snap1', 'dbg_snap2', etc. For the number of such snapshot routines allowed, see the section at the end on the system limits built into **dbg** . To help in writing snapshot routines, the following routine is defined within **dbg** , and made available as an external.

void dbg_print(.....)

This is a routine that takes the same parameters as would be given to a 'printf' statement, but that will write to the **dbg** command stream and/or the **dbg** log stream as appropriate.

A useful technique is to use the power of the C pre-processor to allow you leave the calls to **dbg** in the code, but control whether they actually generate any code. The **debug.h** header file contains some macro definitions that make it easy to control whether **dbg** calls will be generated without the need to modify the source code. If you write the calls to the **dbg** system in the following formats:

```
DBG_INIT();
DBG((paramaters))
DBG_PRINT((parameters))
```

then these statements will generate code if LIBDEBUG is defined, and will generate no code if it is not. You can then control this by using the parameter **-dLIBDEBUG** to the C68 compilation system when you want code generated, and omitting it when you do not.

NOTE. The double brackets are necessary in this case to make the pre-processor handle the variable number of parameters correctly.

RUN TIME INTERFACE

Programs that have had the **dbg** system included are started just as would normally be the case when it is not included.

When the **dbg** system is first entered (normally via the **dbg_init()** call), the user is first asked which channel is to be used for input/output to the **dbg** system. The default is 'stdin/stdout' which is invoked if you merely press ENTER.

It can be particularily advantageous to give a different value if you have another QL networked to the one running the program under test (with networking facilites between them using Toolkit 2). You can then open a channel to the second QL by giving a response such as **n2_con** to this prompt. All **dbg** input/output will then be directed to this second QL. This

means that you will be able to control the **dbg** system without the messages from it corrupting the screen of the program under test.

Once you have answered this first question, you will be asked if you want to write a logfile. The default is none. A log file can be directed either to a disk file, or alternatively direct to a serial port. If a logfile is provided, then all input/output to the **dbg** command channel will also be written to this log file. It is also possible to get trace information output only to a log file, and not to the command stream (see the TMODE command).

Once the initialisation phase has been completed, then **dbg** is invisible to the user until the executing program reaches either a trace point or a stop point. Potentially all calls to **dbg** are both trace points and stop points. Whether a call qualifies as either depends on the setting of the **dbg** global modes (discussed with the commands that set them).

COMMAND DESCRIPTIONS

Once **dbg** has stopped at a stop point, the user interacts with the **dbg** through a conventional command interpreter. All commands conform to the format

<command> [[<arg1>] <arg2>]

in which <command> identifies the command type and <arg1> and <arg2> are either a number or a string literal. Numeric values (paticularily useful for addresses) can be given as either absolute values, or alternatively as **C+value** or **D+value** for addresses relative to the base of the code or data areas respectively. If numeric values start with the **$** symbol they are assumed to be in hexadecimal, and if they start with the **%** symbol the decimal is assumed. If neither is present, then hexadecimal is assumed.

The following is a list of the commands. They are in alphabetical order except for those specific to interfacing with the MALLOC debugging library which are covered at the end. The commands may be typed in either upper or lower case. It is only necessary to type enough of a command to make it unique. In the event of the amount typed not being enough to make the command unique, then the first command encountered of that type willl be assumed.

ADDRESS

This command will display the addresses of the data and code areas for the program under test. The base address will always be given. There will then be a field that varies with the address mode setting. If the address mode is OFF, then the top address of each area is given. If the address mode is ON, then the size of each area will be given.

This information is also displayed when the SHOW command is used to obtain a full summary of the status of the **dbg** system.

AMODE

This command will toggle the display of addresses between absolute format, and as displacements from the base of the code or data areas. Addresses displayed in relative format will be of the form C+value and D+value to represent addresses relative to the code and data areas respectively. Any addresses which fall completely outside both of these areas will always be displayed in absolute format.

CLS

This command simply clears the window associated with command input.

CONTINUE

Return control to the program under execution. You can use GO instead if you wish.

CRC address length

Compute a cyclic redundancy check for the specified block of memory. This command is normally used as a quick way of testing whether a block of instructions or constant data has been changed.

CRCMD mode

Determine how frequently a CRC check will be automatically performed. If the value for a CRC check has changed, then a stop point will be forced. The values allowed for **mode** are

Off All automatic CRC checks off. This is the default.

Stop Do an automatic CRC check at all stop points.

Trace Do an automatic CRC check at all trace points.

All Do an automatic CRC check at al **dbg** calls.

The values for the start address and length will be taken from the last CRC command.

CAUTION: use this option with care as it can cause a severe performance overhead in the program under test, particularily if the length being checked is quite large.

DUMP address length

Print in ASCII and hexadecimal form the specified memory block.

ERRMS

ERRNO

Displays the setting of the global variable **errno** at the time that the current **dbg** call was made.

Note that **dbg** takes special care to preserve the setting of **errno** across **dbg** calls so that programs that make assumptions about when it will change are not affected by whether a **dbg** call has occurred or not.

GO

This is merely an alternative to the CONTINUE command. It returns control to the program uunder test.

GRANULARITY n

Set the trace granularity to **n** . Only every **n** th (otherwise enabled) trace statement will be printed. The argument **n** must always be greater than zero.

This option is normally used when tracing loops.

HELP (or **ENTER** key with no data)

This command will give a summary of the all the commands available in the **dbg** system, and their syntax.

LEVEL n

Sets the trace level to **n** . The value specified here is used by AND'ing it with the detail _index supplied with each **dbg** call. Only if all bits in the detail index have a corresponding bits set in the LEVEL parameter will the **dbg** call be activated.

This means that when designing how you are going to use the detail_index field in the **dbg** calls, it is convenient to consider them as bit fields. Thus you might use bits 0-3 to specify the level of detail, and the remaining bits to specify the type of routine. Such an approach makes it easy to select values for the LEVEL parameter that will activate just the TRACE and STOP points that you are interested in. If adopting this approach, then it is recommended that you specify the detail_index in hex within the program (i.e. 0x101) as this is the format in which you provide values to the LEVEL command.

It is still necessary for the LABEL field on the call to **dbg** to match those given in the trace/stop point settings. The default, however, is to set a trace point pattern of ' ***** ' which would match any label. The default for the level parameter is zero, so if you want to activate trace point displays you must increase this value.

LOCAL n

Print in ASCII and hexadecimal the local variables and parameters associated with the last **n** stack frames (i.e. levels of subroutine call).

The data displayed is in ascending memory address order. This is the reverse to the order in which the variables are defined in the program.

LOG

This command will close the current log file, and then prompt you for the name of a new one. Merely pressing ENTER will cause new log output to be sent to the screen.

POKE address data_byte

This is a very simple memory modification facility that allows you to change a single byte in memory. The 'data_byte' field is the hexadecimal value to be put at that address.

QUIT

Terminate the program under test immediately. Any open files will be closed.

RETRACE

This command is used to redisplay the data that was included as trace information to the current **dbg** call. The data will be displayed on the screen when this command is used even if the screen trace mode is set to OFF.

SHOW

Generate a report that shows the current state of the **dbg** system.

SMODE mode

Set how frequently the stack framepointers and return addresses should automatically be checked as being reasonable (i.e. not pointing outside the stack area). The values for **mode** are

O ff All automatic stack checks off. This is the default.

S top Check at stop points.

T race Check at trace points.

A ll Check at all **dbg** calls.

The framepointers will be checked to ensure that they do not point outside the data area, and also that they are in ascending address order. The return addresses will be checked to ensure that they are within the code area. If either of these checks fail, then an error message will give the details.

SNAP n

Invoke the **n** th User supplied snapshot.

This facility is intended for use when the standard TRACE options are not sufficiently comprehensive. See the section in earlier under 'Programming Interface' for more details.

SSKIP n

The next **n** stop points that would otherwise be enabled are suppressed.

STACK

Give a dump of the framepointers and return addresses for each level of subroutine call. There will be one line for each level of call, with the most recent listed first. The same checks will be made as described under the SMODE command.

STOP n pattern

Create the specified stop point for **dbg** calls that match **pattern** .

In the simplest case patterns are simple string literals that must match exactly the **dbg** call's label (except that case is not relevant).

Wildcards can be used in the pattern. The question mark is used to match any character, including 'no character'. As an example if you typed in the command

```
STOP 1 foo??
```

then this would set step point 1 to the pattern **foo??** which matches

```
foo
foo1
foo12
fooey
```

but does not match the pattern **f00123** . The asterisk, ***** , is used to match any string of characters. Thus the command

```
STOP 1 foo*
```

would set stop point 1 to the pattern **foo*** would match all of the ones in the example above

To cancel a stop point that has already been set, you merely issue a STOP command and leave the **pattern** field blank. Thus the command

```
STOP 1
```

would cancel the current values of stop point 1.

TMODE

This command will toggle the screen trace mode OFF and ON. The default is ON if there is no log file, and OFF is there is a log file. Trace output is normally sent to the **dbg** command stream, and (if activated) to the log file. Setting the screen trace mode to OFF stops the output to the command stream. If a log file has been given, this will continue to receive trace output.

This option is useful when a lot of trace output is being generated, but you do not want to examine it immediately. By only writing it to a log file you avoid excessive data appearing on the screen.

TRACE n pattern

Create the **n** th trace point to match **pattern** . The values allowed in **pattern** are the same as for the STOP command.

A call to **dbg** is treated as a trace point if the pattern matches one of those set using the TRACE command and the level is less than the value set by the LEVEL command. A trace point merely causes the trace data associated with the call to **dbg** to be displayed and does not halt the program. It is, however, possible for the same call to also be treated as a 'stop' call if it also matches one of the conditions for generating a stop.

TSKIP n

The next **n** trace points that would otherwise be enabled are suppressed. This is useful when you want to jump over a known number of trace points (see also the GRANULARITY command).

WATCH n address

Set a watch point at the specified address. Watch points are examined automatically at certain **dbg** calls (according to the mode set by the WATMD command). Whenever an examination reveals that the contents of a watch address have changed, then a stop point is forced.

To disable the **n** th watch point, set the address to zero

WMODE mode

Specify which **dbg** calls are to trigger automatic watch examinations. The options for **mode** are

O ff All watches off. This is the default condition.

S top Examine only at stop points.

T race Examine only at trace points.

A ll Examine at all **dbg** calls.

MALLOC DEBUGGING SYSTEM COMMANDS

The following commands will only have an effect if the **libmalloc_a** library (see LIBMALOC_DOC for details) has been linked into your program.

MFILE filename
This sets/changes the malloc log file. If the filename is ommitted, then the current filename will be displayed. Specifying the special value of **<DEBUG>** sets it to be the same as the **dbg** log file (or the command channel if there is no log file active).

MFATAL value
This sets the malloc_fatal_level. If value is ommitted, then the current setting is displayed instead.

MWARN value
This sets the malloc_warning_level. If value is ommitted, then the current setting is displayed instead.

MCHECK value
This sets the malloc_checking status. If value is ommitted, then the current setting is displayed.

SYSTEM LIMITS

The following are the system limits in the current version of the **dbg** system. If you have the source of this library and you wish to change these limits you need to change the #define statements in the debug_c module and rebuild the library.

Stop Points	3
Trace Points	3
Watch Points	3
User Snapshots	3

C68 Maths library

INTRODUCTION

This is the C68 maths library. Many thanks must go to S.E.Peterson for thw work that he has done in improving the C68 maths library beyond all recognition.

Please note that this library also contains the versions of **printf()** and **scanf()** (and their various close siblings) that have floating point support. This has been done to avoid the need to include the additional floating point code in the vast majority of programs that do not use floating point.

NOTE. If anyone produces more complete documentation for some (or all) of these routines, then please feed them back for inclusion into later releases of C68. Also please feed back any errors that you notice in the documentation.

LIBRARY ROUTINES

The following is a list of all the routines contained in the C68 libm_a Maths library. There is a short summary list followed by a more detailed list organised in alphabetical order.

In a number of sections you will see the name in brackets. This means that a routine of this name exists in that category, but it has not (yet!) been implemented in the C68 libraries.

In some cases the summary lists are split up into a number of levels to indicate the level of portability if you use these routines. In principle the further you go down the following categories, the less portable the code is likely to be.

- ANSI is the general portable C level. This is the one that is defined by the C definition, and the one that provides maximum portability of C source code between disparate systems.
- UNIX is the Unix system call interface. Calls at this level are portable across most Unix sytems. They are also portable to systems which provide libraries to emulate this interface.
- LATTICE is extensions provide with LATTICE C compilers. These options are portable between LATTICE compilers on the various sytems, but not if other compilers are in use. Examples of such compilers under QDOS are QLC and EJC.
- C68 are extension specific to this implementation.

TRIGNOMETRIC FUNCTIONS

ANSI C compatible

acos asin atan atan2

cos sin tan

Lattice C compatible

cot

HYPERBOLIC FUNCTIONS

ANSI C compatible

cosh sinh tanh

Unix compatible

acosh asinh atanh

EXPONENTIAL and LOGARITHMIC FUNCTIONS

ANSI C compatible

exp frexp ldexp log

log10 modf

Unix compatible

expf logff log10f modff

POWER FUNCTIONS

ANSI C compatible

pow sqrt

Unix compatible

hypot powf sqrtf

ROUNDING FUNCTIONS

ANSI C compatible

ceil fabs floor fmod

Unix compatible

ceilf floorf remainder fmodf

MISCELLANEOUS

Unix compatible

copysign matherr

Lattice C compatible

except ecvt fcvt gcvt

C68 compatible

_mult _poly

DATA AREAS

Lattice C compatible

_fperr ----------

double acos(double x)

Returns arccos(x) for -1.0 < =x <= 1.0; the returned value is in the range 0 <= arccos(x) <= pi, otherwise returns -HUGE_VAL.

Function defined in LIBM_acos_c. In ANSI, MS6 and Unix.

---

double acosh (double x)

Returns arccosh(x) for x >=1.0.

Defined in LIBM_acosh_c. Not defined in ANSI; defined in MS6 and Unix.

---

double asin(double x)

Function returns arcsin(x) for -1.0 < =x <= 1.0; the returned value is in the range -pi/2 <= arcsin(x) <= +pi/2, otherwise returns -HUGE_VAL.

Function defined in LIBM_asin_c. In ANSI, MS6 and Unix.

---

double asinh (double x)

Returns arcsinh(x) for x .

Defined in LIBM_asinh_c. Function not defined in ANSI, but defined in MS6 and Unix.

---

double atan2(double y, double x)

Function is a full four quadrant function returning arctan (y/x) where y and x are any two numbers, x != 0. Returned value is in the range -pi <= arctan(y/x) <= +pi.

Defined in LIBM_atan2_c. In ANSI, MS6 and Unix.

---

double atan(double x)

Function returns arctan(x). Returned value is in in the range -pi/2 < arctan(y/x) < +pi/2.

Defined in LIBM_atan_c. In ANSI, MS6 and Unix.

---

double atanh(double x)

Function returns arctanh(x).

Defined in LIBM_atanh_c. Function not defined in ANSI, but defined in MS6 and Unix.

---

double ceil(double x)

float ceilf(float x)

Returns in double format the value of the integer that is the smallest integer greater than or equal to x. Note that ceil(-1.05) is -1.0, while ceil(+1.05) is 2.0.

Defined in LIBM_ceil_c. In ANSI, MS6 and Unix. ceilf defined in LIBM_ceilf_c, but not defined in ANSI or MS6.

---

double copysign (double x, double y)

Returns the value of x with the same algebraic sign as y.

Defined in LIBM_copysign_c. Not defined in ANSI or MS6 but defined in Unix.

---

double cos(double x)

Function returns double cosine of x. There is no restriction on the magnitude of x as it works with the remainder after integral division by 2pi. This does lead, however, to some loss of accuracy with high values of x.

Defined in LIBM_cos_c. Defined in ANSI, MS6 and Unix.

---

double cosh(double x)

Function returns double cosh (hyperbolic cosine) of x.

Defined in LIBM_cosh_c. Defined in ANSI, MS6 and Unix.

---

double cot(double x)

Function returns cotangent of x.

Defined in math.h

---

char *ecvt(double v, int dig, int *decx, int *sign)

Converts double floating point number to a string of characters. v is the double number, dig is the total number of digits (before and after the decimal point) to appear in the string, decx is a pointer to a signed integer value giving the number of characters, left or right, from the beginning of the string to the decimal point. Note that the decimal point is not included in the string returned by the function. sign is a pointer to an integer indicating the sign of the converted number. *sign = 0 if positive, != 0 otherwise.

Defined in fcntl.h? Not defined in ANSI, but defined in MS6 and Unix.

---

double except(int type, char *name, double arg1,

double arg2, double retval)

A lattice facility for simplifying the matherr interface. Sets up the exception vector and processes the action code and return value.

Not implemented in C68 yet. Not defined in ANSI or MS6.

---

double exp(double x)

float expf(float x)

Returns e (base of natural logarithms) raised to the power x.

Defined in LIBM_exp_c. double exp defined in ANSI, MS6 and Unix; float expf not defined in ANSI or MS6.

---

double fabs(double d)

float fabsf(float d)

fabs returns the absolute value of a double floating point number. See also abs().

Defined in LIBM_fabs_c. Defined in ANSI, MS6 and Unix. fabsf returns the float value of a float number; it is not defined in ANSI or MS6.

---

char *fcvt(double v, int dec, int *decx, int *sign)

Converts double floating point number to a string of characters. v is the double number, dig is the number of digits to be stored after the decimal

point (all digits prior to the decimal point will be included in the string), decx is a pointer to a signed integer value giving the number of characters, left or right, from the beginning of the string to the decimal point. Note that the decimal point is not included in the string returned by the function. sign is a pointer to an integer indicating the sign of the converted number. *sign = 0 if positive, != 0 otherwise. See also *gcvt.

Defined in fcntl_h. Not defined in ANSI, but defined in MS6 and Unix.

---

double floor(double x)

float floorf (float x)

Returns a double (floating point) value representing the largest integer that is smaller than x, e.g. floor(6.5) = 6.0 and floor(-6.5) = -7.0 . double floor defined in LIBM_floor_c.

Defined in ANSI, MS6 and Unix. float floorf defined in LIBM_floorf_c; it is not defined in ANSI or MS6. ----------

---

double fmod(double x, double y)

float fmodf(float x, float y)

Returns floating point remainder on divison of x by y. More precisely, the number returned (f) has the saem sign as x, such that x=iy + f for some integer i, and |f| < |y|. Sign of remainder is same as sign of x and abs(r) < abs(y). Useful e.g. for reducing large angles to the range 0 to 2pi. Defined in LIBM_fmod_c. Similar function to remainder(x,y)(see below).

Defined in ANSI, MS6 and Unix.

---

double frexp(double v, int *xp)

The function breaks down v into a mantissa m and exponent n such that 0.5 <= abs(m) < 1.0 and v = m * 2^n. The function returns the value of the mantissa with the same sign as v. The integer exponent n is stored at the location pointed to by xp.

Defined in stdlib_c. Defined in ANSI, MS6 and Unix.

NOTE. This routine is always implementation specific as it has to know the underlying representation of floating point.

---

double gamma(double x)

Returns the value of gamma(x), the generalized factorial. Note that it is very easy to exceed the numeric range of the machine with relatively small values of x, e.g. gamma(100) = 9.3E155, hence it is common to work with the logarithm of gamma(x), defined in lgamma(x) (see below). Implemented only for positive values of x.

Defined in LIBM_gamma_c. Not defined in ANSI or MS6, but defined in Unix.

---

char *gcvt(double v, int dig, char *buf)

Converts a double f.p. v to a string stored in a buffer which has a terminating \0. It attempts to produce dig significant figures in decimal format, failing which it will attempt to produce dig significant figures in exponential format. The function returns a pointer to the string of digits. Defined in stdlib.h ? Not defined in ANSI, but defined in MS6 and Unix.

---

double hypot(double x, double y)

Returns the length of the hypotenuse of a right triangle with sides x and y.

Defined in LIBM_hypot_c. Not defined in ANSI, but defined in MS6 and Unix.

---

double ldexp(double v, int x)

Returns the double value of v * 2^x. This is the inverse of frexp function.

Defined in math.h. In ANSI, MS6 and Unix.

NOTE. This routine is always machine specific as it has to know the underlying representation of floating point numbers.

---

double lgamma(double x)

Returns the logarithm of the gamma function (generalised factorial.) for x > 0.0 . Not defined in ANSI or MS6, but defined in Unix.

---

double log10(double x)

float log10f(float x)

Returns the logarithm of x to base 10. double defined in LIBM_log10_c.

Defined in ANSI, MS6 and Unix. float log10f defined in LIBM_log10f_c; it is not defined in ANSI or MS6.

---

double log(double x)

float logf(float x)

Returns the natural logarithm of x. double defined in LIBM_log_c; it is defined in ANSI, MS6 and Unix. float logf is defined in LIBM_logf_; it is not defined in ANSI or MS6.

---

int matherr(struct exception *x)

Returns 0 to indicate an error and non-zero for succesful corrective action.

Defined in math.h. ANSI has said that this is now obsolete. Defined in MS6 and Unix (if ANSI option disabled). Not yet implemented.

---

double modf(double y, double *p)

float modff(float x, flaot y)

Breaks down a double floating point value into integer and fractional parts. The function returns the signed fractional portion. The pointer p points to the address of a double floating point variable containing the integer part.

Defined in ANSI, MS6 and Unix.

NOTE. This routine is always machine specific as it has to know the underlying representation of floating point numbers.

---

double pow (double x, double y)

float powf(float x, float y)

Returns the value of x raised to the power y.

Defined in LIBM_pow_c. Defined in ANSI, MS6 and Unix.

---

double remainder(double x, double y)

Returns double remainder on divison of x by y. More precisely, it returns the value r = x - yn, where n is the integer nearest the exact value x/y. Whenever |n - xy| = o.5, then n is even. Very similar to fmod(x,y) except for rounding under boundary conditions.

remainder is not defined in unix, but not in ANSI or MS6.

---

double rint (double x)

Function returns nearest integer value to its floating point argument x as a floating point number. The returned value is rounded according to the machines (current) rounding mode. If round-to-nearest (the default) is set and the difference between the function argument and the rounded result is exactly 0.5, then the result will be rounded to the nearest even integer.

Defined in math.h

---

double sin(double x)

Returns sin(x). Works on remainder after division by 2pi, hence there is some loss of accuracy with large values of x.

Defined in LIBM_sin_c. Defined in ANSI, MS6 and Unix.

---

double sinh(double x)

Returns value of hypberbolic sine of x.

Defined in LIBM_sinh_c. Defined in ANSI, MS6 and Unix.

---

double sqrt(double x)

float sqrtf(float x)

Returns square root of argument.

Defined in LIBM_sqrt_c. double defined in ANSI, MS6 and Unix; float not defined in ANSI or MS6.

---

double tan(double x)

Returns value of tangent of x. Operates on remainder after dividing argument by 2pi, hence there is some loss of accuracy for large values of the argument.

Defined in LIBM_tan_c. Defined in ANSI, MS6 and Unix.

---

double tanh(double x)

Returns hyperbolic tangent of x.

Defined in LIBM_tanh_c. Defined in ANSI, MS6 and Unix.

Defined in LIBM_tanh_c. Defined in ANSI, MS6 and Unix.

INTERNAL ROUTINES

The following routines are implemented as support functions for other routines in the C68 maths library. Use of these routines is definitely non-portable.

---

double _mult(double x, double y)

A "safe" multiplication routine which returns x*y if in range, otherwise HUGE_VAL with proper sign if not. Introduced as a precaution because of the limited numeric range of MFFPF.

Defined in LIBM__mult_c. Not defined in ANSI or MS6.

---

double _poly (double x, double *coeff, int ncoeff)

Returns the value of a polynomial for the argument x using Horner's method. *coeff is the array of coefficients of the polynomial beginning with a0 (the constant term). Note that all coefficients must be supplied even when they are zero. ncoeff is the total number of coefficients (which is one more than the index of the highest order coefficient and equals the order of the ploynomial).

Defined in LIBM__poly_c; Internal routine to C68 library.

VARIABLES

---

extern int _fperr

This is set if the underlying routines which perform floating point arithmetic detect an error.

Defined in math.h

QDOS System Call Interface

This section of the C68 library documentation covers those routines in the C68 standard library that provide access to the QDOS operating system interfaces.

All of the calls in this part of the library map directly onto the QDOS System Calls available to Assembler (machine code) programmers. It is therefore useful to have access to documentation covering the Assembler level interface to QDOS if you want more details on how many of these calls work.

You do not ever need to tell the linker explicitly that you want to include routines defined in this document. These routines are imbedded in the LIBC_A library which is automatically included at the end of the link by the LD linker. You must always, however have the statement

#include <qdos.h>

in any program or module that makes use of the routines in this library.

It is worth noting that all the calls defined here also work on the SMS family of operating systems. However in that case they traditionally have alternative names. If you wish to find the functions listed and described under their SMS names, then refer to the LIBSMS_DOC file.

REFERENCE MATERIAL

The reference books listed below were used in preparing material for inclusion in this library:

"QL Technical Guide" by David Karlin and Tony Tebby

"QL Advanced User Guide" by Adrian Dickens

"QDOS Reference Manual" as published by Jochen Merz

---

int c_extop (chanid_t channel, timeout_t timeout,

int (*func), int number_of_params, ...)

This routine allows a routine to be called to do an extended operation on a QDOS channel. The parameters are passed in a way that is compatible with this routine being written in C (c.f. sd_extop()/iow_xtop() for assembler only routines).

The C routine will be called in supervisor mode, with the parameters specified by ... above passed to it on the stack. Each parameter is assumed to be no larger than 4 bytes in size (i.e. no structures are to be passed on the stack). Note also that due to a bug in QDOS, it seems to hang if the routine does not return zero in D0. Therefore, if it is desired to pass an error code back to the application program it must be done indirectly via one of the parameters.

---

char *cn_date(char *asciidate, time_t qldate)

Converts a date from internal QL format into an ASCII string in the format "YYYY mmm dd hh:mm:ss". The asciidate parameter must point to a buffer of at least 25 characters in length to hold the return data. The buffer returned is in QL string format - which is a 2 byte length field, followed by the data (NULL terminated for convenience to C programmers). The return value is the address of the start of the text.

Note that if you intend to access the length field of the buffer you MUST ensure that it starts on an even address - preferably by defining it using the QLSTR_DEF macro to define the buffer.

---

char *void cn_day(char *asciiday, time_t qldate)

Returns the 3 character day of the week given a date in QL internal format. The asciidate parameter must point to a buffer of at least 7 characters in length to hold the return data. The buffer returned is in QL string format - which is a 2 byte length field, followed by the data (NULL terminated for convenience to C programmers). The return value is the address of the start of the text.

Note that if you intend to access the length field of the buffer you MUST ensure that it starts on an even address - preferably by defining it using the QLSTR_DEF macro to define the buffer.

---

void cn_ftod (char * target, char * value)

Routine to convert a QDOS floating point value into a decimal character ASCII string.

---

void cn_itobb (char * target, char * value)

Routine to convert a byte into a 8 character ASCII string of binary.

---

void cn_itobl (char * target, long * value)

Routine to convert a long integer into a 32 character ASCII string of binary.

---

void cn_itobw (char * target, short * value)

Routine to convert a short integer (word) into a 16 character ASCII string of binary.

---

void cn_itod (char * target, short * value)

Routine to convert a short integer into a decimal ASCII string.

---

void cn_itohb (char * target, char * value)

Routine to convert a byte into a 2 character ASCII hex string.

---

void cn_itohl (char * target, long * value)

Routine to convert a long integer into a 8 character ASCII hex string.

---

void cn_itohw (char * target, short * value)

Routine to convert a short integer (word) into a 4 character ASCII hex string.

---

int fs_check(chanid_t channel, timeout_t timeout)

QDOS routine to check for pending operations on a file. Returns 0 if operations have completed, QDOS error code (typically -1 for Not complete) if they haven't.

---

int fs_date(chanid_t chan, timeout_t timeout, int type,
long * sr_date)

type = 0 Access update date of file,

= 2 Access backup date.

*sr_date = -1 Read requested date (returned from call in *sr_date)

= 0 Set requested date to current date.

else Set requested date to date given in *sr_date.

Read/Set update or backup dates. Available on Miracle Systems hard disk, ST/QL sytems and SMS systems. The date set/read is returned in *sr_date. Returns QDOS error code.

---

int fs_flush(chanid_t channel, timeout_t timeout)

int fs_flush(chanid_t channel, timeout_t timeout)

QDOS routine to flush all buffers on a file. Returns QDOS error codes.

---

int fs_headr(chanid_t chan, timeout_t timeout,
void * buf, short buflen)

QDOS routine to read a file header. Returns length read on success, QDOS error code (which is negative) on failure.

---

int fs_heads(chanid_t chan, timeout_t timeout,
void * buf, short buflen)

QDOS routine to save a file header. Returns length written on success, QDOS error code (which is negative) on failure. You must have opened the file with a mode that allows writing for this call to be successful.

---

long fs_load(chanid_t channel, char * buf,
unsigned long len)

Routine to load a complete file. Returns length loaded on success, QDOS error code (which is negative) on error.

---

int fs_mdinf(chanid_t chan, timeout_t timeout,char * medname,
short * unused_secs, short * goodsecs)

Routine to get media information. Returns 10 character name of media (N.B. not NULL terminated), number of unused_sectors, and number of good sectors. Returns QDOS error code.

---

int fs_mkdir(chanid_t channel)

Make the file specified by the QDOS channel into a directory. Requires support for Level 2 filing system (e.g. Miracle hard Disk, ST/QL or SMS systems). Returns QDOS error code.

---

long fs_pos(chanid_t chan, long pos, int mode)

QDOS equivalent to C seek() routine to seek to a point in a file (no timeout as it's always -1). The parameter 'mode' can have the following values:

0 absolute

1 relative to current position

2 relative to EOF.

Returns new position on success, and QDOS error code (which is negative) on failure.

---

long fs_posab(chanid_t chan, timeout_t timeout,
unsigned long * pos)

Routine to seek to an absolute point in a file. The new file position is returned via the 'pos' parameter. Returns QDOS error code.

---

long fs_posre(chanid_t chan, timeout_t timeout, long * pos)

Routine to seek to a point in a file relative to the current position. The new file position is returned via the 'pos' parameter. Returns QDOS error code.

---

int fs_rename(char * old, char * new)

Routine to rename a file. Uses C strings. Calls toolkit 2 routine. Returns QDOS error code.

---

int fs_save(chanid_t channel, char * buf, unsigned long len)

Routine to save a complete file to a channel. Returns length saved on success, QDOS error code (which is negative) on failure.

---

int fs_trunc(chanid_t channel, timeout_t timeout)

Routine to truncate a file at the current byte position. This call may not be available on very basic QL systems (unless Toolkit 2 present) but all other types of system can be expected to support it. Returns QDOS error code.

---

int fs_vers(chanid_t channel, timeout_t timeout, long * key)

Set/Read a file version number. Only available on systems that support version2 filing systems (such as Miracle hard disk, ST/QL and SMS systems). The action is defined as follows:

*key = -1 Return version number in *key.

= 0 Keep old version number when file closed (return it on *key)

+ve and < 65536 Set version number to given number.

Returns QDOS error code.

---

int fs_xinf(chanid_t channel, timeout_t timeout,
struct ext_mdinf * fsinf)

Get extended file system info. Only available on systems that support version2 filing system (such as Miracle hard disk, ST/QL and SMS systems). Requested data is returned in struct ext_mdinf (defined in qdos.h) on success. Returns QDOS error code

---

int io_close (chanid_t channel)

Closes a channel. Returns QDOS error code.

---

int io_delete(char *name)

Routine to delete a file. Uses C strings. Returns QDOS error code.

---

int io_edlin(chanid_t channel, timeout_t timeout,
char **cptr, int bufsize,
int current_offset, int *current_linelen);

Routine to do edited line read call. Returns QDOS error code.

---

int io_fbyte(chanid_t channel, timeout_t timeout,
char *char_pointer)

Routine to read 1 byte. Returns QDOS error code.

---

int io_fdate(lchanid_t chan, timeout_t timeout, int type,
unsigned long *sr_date)

Obsolete form - should now use fs_date() (or even better, the SMS name iof_date())instead.

---

int io_fline(chanid_t channel, timeout_t timeout,
void *buf, short length)

Routine to read a linefeed terminated string of bytes. Returns length read on success, QDOS error code (which is negative) on failure.

---

int io_format(char *device, short *totsecs, short *goodsecs)

Routine to format a medium, uses C string name. Returns total and good sector count. Returns QDOS error code.

---

int io_fstrg(chanid_t channel, timeout_t timeout,
void *buf, short length)

Routine to fetch a string of bytes. Returns length read on success, or QDOS error code (which is negative) on failure. The amount read can be less than the amount requested. This would normally caused by an end-of-file or timeout condition occurring during the read.

---

int io_fvers(chanid_t channel, timeout_t timeout, long *key)

Obsolete form - should now use fs_vers() instead.

---

int io_fxinf(chanid_t channel, timeout_t timeout,
struct ext_mdinf *fsinf)

Obsolete form - should now use fs_xinf() instead.

---

int io_mkdir(chanid_t channel)

Obsolete form - should now use fs_mkdir() instead.

---

chanid_t io_open(char *name, int mode)

Routine to open a file. Uses C strings. Returns channel id or QDOS error code (which is negative).

---

int io_pend(chanid_t chan, timeout_t timeout)

Routine to test for any pending input on a channel, returns 0 if data is to be read, else -1 (not complete).

---

int io_qeof(char * queue_pointer)

Insert and EOF (end-of-file) marker into a queue. Returns QDOS error code (if any).

---

int io_qin (char * queue_pointer, int byte_to_insert)

QDOS routine to insert a byte in a queue. Returns the QDOS error code (if any).

---

int io_qout (char * queue_pointer, char * next_byte)

Remove a byte from a queue. Returns the QDOS error code (if any).

---

void io_qset(char * queue_pointer, long queue_length)

Routine to set up a queue.

---

int io_qtest(char * queue_pointer, char * next_byte,
long * free_space)

Test the status of a queue. The variables whose addresses are passed as parameters are updated to the free space in the queue, and (if there is data in the queue) the value of the next byte is returned (although the byte is not removed from the queue). The QDOS error code is returned.

---

int io_rename(char *old, char *new)

Obsolete form - should now use fs_rename() instead.

---

int io_sbyte(long chan, timeout_t timeout, unsigned char ch)

Routine to output char ch to channel. Returns QDOS error code.

---

int io_serio(chanid_t channel_id, timeout_t timeout,
int routine_number, long * D1, long * D2,
char ** A1, char * routine_array[4])

General serial IO handling routine. This routine is used when the io_serq() routine is not sufficient. The values passed as the parameters 'D1', 'D2' and 'A1' are pointers to the values to be put into the registers D1, D2 and A1 respectively. These values may be changed by this routine. The 'routine_array' is an array of at least 4 elements, the first three of which contain the addresses of the routines for testing pending input, fetching a byte and sending a byte. The fourth element will be used as workspace, and thus corrupted by this call.

---

int io_serq (chanid_t channel_id, timetout_t timeout,
int routine_number, long * D1, long * D2,
char ** A1)

Serial IO Direct Queue handling routine. The values passed as the parameters 'D1', 'D2' and 'A1' are pointers to the values to be put into the registers D1, D2 and A1 respectively. These values may be changed by this routine.

---

int io_sstrg(chanid_t channel, timeout_t timeout,
void *buf, short length)

Routine to write a string of bytes. Returns length written on success, and a QDOS error code (which is negative) on failure. The amount written can be less than the amount requested. This would normally be caused by a timeout condition occuring during the write.

---

int io_trunc (chanid_t channel, timeout_t timeout)

Obsolete form - should now use fs_trunc() instead.

---

int iop_outl (chanid_t channel, timeout_t timeout,
short, short, short, void *)

This is the call that sets the outline window for a Pointer Environment. It is included in this library as it is the one call that need to be issued to make a program that is not otherwise aware of the pointer environment function correctly in that environment.

For more details refer to the LIBQPTR_DOC file provided as part of the QPTR library.

Note that the default console initialisation routines supplied with C68 will automatically issue a call to set the window outline to the size as defined in the ' condetails' global variable (for more information see

LIBC68_DOC).

---

char * mm_alchp(long size, long *sizegot)

Routine to allocate memory from common heap. It is passed the requested size and returns address of area allocated (or a QDOS error code on failure). The area will always be allocated with the current job as the owner. If you are not interested in the true size obtained, then set 'sizegot' to NULL. Otherwise set it to the address of a variable that will be set to contain the actual size obtained (Note that even if the call succeeds this may not be the same as the size requested, as the amount requested is often rounded up by QDOS. It is recommended that you use **mt_alchp()** in preference to **mm_alchp()** unless you are sure you know what you are doing.

WARNING

The size requested must allow for the QDOS heap header, and the address returned is the start of the area allocated - not the useable area. This is in contrast to the **mt_alchp()** call for which the user does not have to worry about the QDOS heap header.

---

char *mm_alloc(char **ptr, long *len)

QDOS routine to allocate a user area from an allocated area of common heap. 'ptr' is a pointer to a pointer to free space, len is the length requested to put in the user heap, and returns as the length actually allocated. Returns the address of the area allocated on success, and the QDOS error code on failure.

---

void mm_lnkfr(char *area, char **ptr, long len)

QDOS routine to link an area back into a user heap area. Given area to link in, pointer to pointer to free space, and length to link in. This call is also used to set up a user heap.

---

void mm_rechp(char *area)

QDOS routine to free an area of common heap previously allocated via **mm_alchp()** . Returns no errors. It always succeeds unless the parameter points to an invalid address, in which case the machine nearly always crashes!

---

void mt_aclck(long ql_time)

Routine to adjust the clock by ql_time seconds.

---

int mt_activ(long jobid, unsigned char priority,
timeout_t timeout)

Routine to start a activate a job with a given priority. There are two valid values for the timeout, 0 and -1. Execution of the current job will continue if the timeout is set to zero, and the QDOS error code for this call returned. If the timeout is -1 then the current job is suspended until the activated Job has finished. This call will then return the error code from that Job.

---

char * mt_alchp(long size, long * sizegot, long jobid)

Routine to allocate memory from common heap. Is passed requested size, plus job id which is to own the heap. Returns address of area allocated, or a QDOS error code on failure.

Note that even if the call succeeds the amount of memory actually allocated will not be the same as the size requested, as the amount requested is rounded up to the nearest 16 bytes and then the length of the common heap header is added on to it. If you are not interested in the true size obtained, then set 'sizegot' to NULL. Otherwise set it to the address of a variable that will be set to contain the actual size obtained.

---

void * mt_alloc(char **ptr, long *len)

Routine to allocate a user area from an allocated area of common heap. 'ptr' is a pointer to a pointer to free space, len is the length requested to put in the user heap, and returns as the length actually allocated. Returns the address of the area allocated on success, and the QDOS error code on failure.

---

void * mt_alres(long size)

Routine to allocate memory from resident procedure area. Returns address of area allocated, or a QDOS error code on failure. On standard QDOS systems this call will always fail if called while any program except SuperBasic is executing. Most later sysems and those fitted with Minerva ROMs do not

suffer from this limitation.

---

void mt_baud(int rate)

Routine to set the baud rate for both serial ports.

---

jobid_t mt_cjob(long codespace, long dataspace,
char *start_address, jobid_t owner,
char **job_address)

Routine to create another job in the transient program area, given size of new jobs code, data the start address of the new job, and its owner. Returns either positive job id of new job, or QDOS error code. Also returns address of newly created job in last parameter.

---

void mt_dmode(short *s_mode, short *d_type)

Routine to set/read display mode.

```
 *s_mode  = 4   for mode 4,
          = 8   for mode 8,
          = -1  for read
 *d_type  = 0   for monitor mode,
          = 1   for TV mode,
          = -1  for read
```

Notes:

1. Other values are available for use in these parameters on Minerva ssytems - refer to the Minerva documentation for details
2. There is a bug in some QL roms that corrupts the return d_type when it is read.

---

long mt_free ()

Routine to find largest contiguous area available for loading a program. This is normally also a good indicator of the total free memory in the machine.

---

int mt_frjob(jobid_t jobid, int error_code)

Routine to force remove a job, giving an error code for it to return. Returns QDOS error code (if we are not removing the current job).

---

jobid_t mt_inf(char **system_variables, long *version_code)

Routine to get the address of the system variables and the current operating system version code. The version code is actually returned as 4 bytes in the form x.xx. Returns job id of current job.

---

int mt_ipcom(char *param_list)

Routine to send a command to the 8049 second processor. Uses INTEL byte format (low byte first). Returns value returned by 8049.

---

int mt_jinf(jobid_t *jobid, jobid_t *topjob,
long *job_priority, char **job_address)

Get information on a job within a job tree. Passed the jobid you want information on and the current top of the job tree you are looking at (with the first call set *topjob = *jobid). It is designed to be called repeatedly without changing jobid and topjob until *jobid == 0. Returns:

0 OK with
'job_address' contains address of job
'jobp' contains job priority in least significant byte, and if the job is suspended the most significant byte is negative.
'jobid' and 'topjob' are changed to those of the next job in the tree.
-ve QDOS error code.

---

void mt_lnkfr(char *area, char **ptr, long len)

Routine to link an area back into a user heap area. Given area to link in, pointer to pointer to free space, and length to link in. This call is also used to set up a user heap.

---

void mt_lxint(QL_LINK_t * lnk)

Link in external interrupt handler

Link in external interrupt handler

void mt_rxint(QL_LINK_t * lnk)

Unlink external interrupt handler

void mt_lpoll(QL_LINK_t * lnk)

Link in polled task handler

void mt_rpoll(QL_LINK_t * lnk)

Unlink polled task handler

void mt_lschd(QL_LINK_t * lnk)

Link in scheduler list handler

void mt_rschd(QL_LINK_t * lnk)

Unlink scheduler list handler

void mt_liod(QLD_LINK_t * lnk)

Link in simple I/O device handler

void mt_riod(QLD_LINK_t * lnk)

Unlink simple I/O device handler

void mt_ldd(QLDDEV_LINK_t * lnk)

Link in directory I/O device handler

void mt_rdd(QLDDEV_LINK_t * lnk)

Unlink directory I/O device handler

The QL_LINK_t, QLD_LINK_t and QLDDEV_LINK_t structures are defined in sys/qlib.h

---

int mt_prior(long jobid, int new_priority)

Routine to set the priority of a job. Sets current jobs priority if jobid = -1. Returns old priority of this job or a QDOS error code.

---

long mt_rclck()

Routine to read clock. Returns time in seconds from Jan 1 1961.

---

void mt_rechp(char *area)

Routine to free an area of common heap previously allocated. Returns no errors.

WARNING The way this call is implemented in QDOS and SMS is such that it either succesds or crashes the sytem if given an invalid area address. Do not therefore try and call it twice for the same area or call it for an area not allocated vi **mt_alchp()** call.

JOBHEADER_t * mt_reljb(jobId_t jobid)

Routine to release a suspended job, sets _oserr, returns address of job header (the JOBHEADER_t structure is defined in sys/qlib.h) or QDOS error code.

---

int mt_reres(char *area)

Routine to free an area of the resident procedure area previously allocated. Returns QDOS error code. On QDO systems, will always fail if called when aby program except SuperBasic is running.

---

int mt_rjob(jobid_t jobid, int error_code)

Routine to remove a suspended job, giving an error code for it to return. Returns QDOS error code.

---

void mt_sclck(long ql_time)

Routine to set the clock.

---

int mt_shrink(char *block, long newsize)

Routine to shrink an area of QDOS allocated common heap. This is used when you have grabbed an area of common heap and realise you do not need all of it. Rather than freeing all of it then re-allocating (by which time another job may have grabbed the space) you can use this call to release the top part of it that you do not need. newsize MUST be less than the size originally allocated or this call can fail badly, after the call the allocated block will be only newsize bytes long (not including common heap header), the higher portion of it will have been given back to QDOS and placed on the free list. Returns a QDOS error code.

---

int mt_susjb(jobid_t jobid, int number, char *zero)

Routine to suspend a job for a number of 50Hz (or 60Hz if an American QL)

clock ticks. char *zero is an address of a byte to set to zero on release of the job if required. If this is not required pass NULL in place of 'char *zero'. If number = -1 then the job is suspended indefinitely. Returns a QDOS error code.

---

int mt_trans (char * trans_table, char * msg_table)

Routine to set the translate table and message table. This routine will not work on QL systems with ROMS that are of version JS or earlier. Returns QDOS error code.

---

int mt_trapv(QLVECTABLE_t * table, long jobid)

Routine to change the exception vector table for a particular job. The QLVECTABLE_t structure is defined in sys/qlib.h Returns QDOS error code.

---

int sd_arc(chanid_t channel, timeout_t timeout,
double x_start, double y_start,
double x_end, double y_end, double angle)

Routine to draw an arc using graphics coordinates. sd_arc uses C double precision floating point coordinates (cf. sd_iarc). Returns QDOS error code.

---

int sd_bordr(chanid_t channel, timeout_t timeout,
unsigned char colour, short width)

Routine to redifine a window border with new colour and width. Returns QDOS error codes.

---

int sd_chenq(chanid_t channel, timeout_t, QLRECT_t *rect)

Routine to read a window size in characters. On success 'rect' is set to details of answer. Returns QDOS error code.

---

int sd_clear(chanid_t channel, timeout_t timeout)

Routine to clear entire window. Returns QDOS error code

---

int sd_clrbt(chanid_t channel, timeout_t timeout)

Routine to clear area of window below cursor line. Returns QDOS error code.

---

int sd_clrln(chanid_t channel, timeout_t timeout)

Routine to clear all of cursor line. Returns QDOS error code.

---

int sd_clrrt(chanid_t channel, timeout_t timeout)

Routine to clear cursor line, to right of cursor position (including cursor). Returns QDOS error code.

---

int sd_clrtp(chanid_t channel, timeout_t timeout)

Routine to clear area of window above cursor line. Returns QDOS error code.

---

int sd_cure(chanid_t chan, timeout_t timeout)

Routine to enables cursor on screen channel. Returns QDOS error code.

---

int sd_curs(chanid_t chan, timeout_t timeout)

Routine to suppress cursor on screen channel. Returns QDOS error code.

---

int sd_donl(chanid_t channel, timeout_t timeout)

Routine to flush any pending newlines on a window channel. Returns QDOS error code.

---

int sd_elipse(chanid_t channel, timeout_t timeout,
double x_centre, double y_centre,
double eccentricity, double radius,
double angle_of_rotation)

Routine to draw a circle or ellipse using graphics coordinates. sd_elipse uses C double precision floating point coordinates (cf. sd_ielipse). Returns QDOS error code.

---

int sd_extop(chanid_t channel,timeout_t timeout,int (*rtn)(),
long paramd1, long paramd2, void *parama1)

Routine to do extended operation on screen channel. Passed address of routine to call and parameters for d1, d2 and a1. Returns QDOS error code. See also c_extop().

NOTE. Due to a bug in QDOS, it appears that D0 must always be zero on exiting the rtn() function. Any error code therefore needs to be passed back indirectly via one of the other parameters.

---

int sd_fill(chanid_t channel, timeout_t timeout,
colour_t colour, QLRECT_t * rect)

Routine to plot a rectangular block of a certain colour. Can be used to draw very fast horizontal and vertical lines. Returns QDOS error code.

---

int sd_flood(chanid_t channel, timeout_t timeout, int onoff)

Routine to set flood fill mode on or off. Returns QDOS error code.

---

int sd_fount(chanid_t channel, timeout_t timeout,
char *font1, char *font2)

Routine to set normal and alternative character font in a window. Passed pointers to two font definitions (format as described in QDOS manuals). Returns QDOS error code.

---

int sd_gcur(chanid_t channel, timeout_t timeout,
double vert_offset, double horiz_offset,
double x_pos, double y_pos)

Routine to set the graphics text cursor. sd_gcur uses C double precision floating point coordinates (cf. sd_igcur). Returns QDOS error code.

---

int sd_iarc(chanid_t channel, timeout_t timeout,
double x_start, double y_start,
double x_end, double y_end, double angle)

Routines to draw an arc using graphics coordinates. sd_iarc takes integer coordinates (c.f. sd_arc) Returns QDOS error code.

---

int sd_ielipse(chanid_t channel, timeout_t timeout,
int x_centre, int y_centre, int eccentricity, int radius, int angle_of_rotation)

Routine to draw a circle or ellipse using graphics coordinates. sd_ielipse uses integer coordinates (cf. sd_elipse). Returns QDOS error code.

---

int sd_igcur(chanid_t channel, timeout_t timeout,
int vert_offset, int horiz_offset,
int x_pos, int y_pos)

Routine to set the graphics text cursor. sd_igcur uses integer coordinates (cf. sd_gcur). Returns QDOS error code.

---

int sd_iline(chanid_t channel, timeout_t timeout,
int x_start,int y_start, int x_end, int y_end)

Routine to draw a line with graphics coordinates. sd_iline takes integer coordinates (cf. sd_line). Returns QDOS error code.

---

int sd_ipoint(chanid_t channel, timeout_t timeout,
int x, int y)

Routine to plot a point using graphics coordinates. sd_ipoint takes integer coordinates (cf. sd_point). Returns QDOS error code.

---

int sd_iscale(chanid_t channel, timeout_t timeout,
int scale, int x_origin, int y_origin)

Routine to change a windows graphics origin and scale. sd_iscale uses integer coordinates (cf. sd_scale). Returns QDOS error code.

---

int sd_line(chanid_t channel, timeout_t timeout,
double x_start, double y_start,
double x_end, double y_end)

Routine to draw a line with graphics coordinates. sd_line uses C double precision floating point coordinates (cf. sd_iline). Returns QDOS error code. ---

int sd_ncol(chanid_t channel, timeout_t timeout)

Routine to move cursor right one column. Returns QDOS error code.

Routine to move cursor right one column. Returns QDOS error code.

---

int sd_nl(chanid_t channel, timeout_t timeout)

Routine to move the cursor to start of next line. Returns QDOS error code.

---

int sd_nrow(chanid_t channel, timeout_t timeout)

Routine to move cursor down one row. Returns QDOS error code.

---

int sd_pan(chanid_t channel, timeout_t timeout, int ampix)

Routine to pan window left or right. ampix < 0 means pan left, ampix > 0 means pan right. Returns QDOS error code.

---

int sd_panln(chanid_t channel,timeout_t timeout, int ampix)

Routine to pan cursor line left or right. ampix < 0 means pan left, ampix > 0 means pan right. Returns QDOS error code.

---

int sd_panrt(chanid_t channel,timeout_t timeout, int ampix)

Routine to pan right of cursor line left or right (includes character at cursor position). ampix < 0 means pan left, ampix > 0 means pan right. Returns QDOS errors code.

---

int sd_pcol(chanid_t channel, timeout_t timeout)

Routine to move cursor left one column. Returns QDOS error code.

---

int sd_pixp(chanid_t channel, timeout_t timeout,
short x_pos, short y_pos)

Routine to reposition the cursor to an x, y pixel position in a window. Returns QDOS error code.

---

int sd_point(chanid_t channel, timeout_t timeout,
double x, double y)

Routine to plot a point using graphics coordinates. sd_point takes C double precision floating point coordinates (cf. sd_ipoint). Returns QDOS error code.

---

int sd_pos(chanid_t channel, timeout_t timeout,
short x_pos, short y_pos)

Routine to reposition the cursor to an x, y character position in a window. Returns QDOS error code.

---

int sd_prow(chanid_t channel, timeout_t timeout)

Routine to move cursor up one row. Returns QDOS error code.

---

int sd_pxenq(chanid_t channel, timeout_t timeout,
QLRECT_t * rect)

Routine to read a window size in pixels. Returns size in the QLRECT_t structure (defined in sys/qlib.h). Returns QDOS error code.

---

int sd_recol(chanid_t channel, timeout_t timeout,
char *colourlist)

Routine to recolour a window. Done in software and very slow. colourlist points to eight characters containing new colours for eight possible QL colours. Returns QDOS error code.

---

int sd_scale(chanid_t channel, timeout_t timeout,
double scale,double x_origin, double y_origin)

Routine to change a window's graphics origin and scale. sd_scale uses C double precision floating point coordinates (cf. sd_iscale). Returns QDOS error code.

---

int sd_scrbt(chanid_t channel,timeout_t timeout, int ampix)

Routine to scroll window below cursor line up or down. ampix < 0 means scroll down, ampix > 0 means scroll up. Returns QDOS error code.

---

int sd_scrol(chanid_t channel,timeout_t timeout, int ampix)

Routine to scroll entire window up or down. ampix < 0 means scroll down, ampix > 0 means scroll up. Returns QDOS error code.

---

int sd_scrtp(chanid_t channel,timeout_t timeout, int ampix)

Routine to scroll window above cursor line up or down.

ampix < 0 means scroll down, ampix > 0 means scroll up. Returns QDOS error code.

---

int sd_setfl(long chan, timeout_t timeout, int onoff)

Routine to set flash mode on or off (only works in 8 colour mode). Returns QDOS error code.

---

int sd_setin(long chan, timeout_t timeout, int colour)

Routine to set ink colour. Colour value (0-7) dependent on mode. Returns QDOS error code.

---

int sd_setmd(chanid_t channel, timeout_t timeout, int mode)

Routine to set type of drawing mode (DM_XOR, DM_OVER, DM_OR). Returns QDOS error code.

---

int sd_setpa(long chan, timeout_t timeout, int colour)

Routine to set paper colour. Colour value (0-7) dependent on mode. Returns QDOS errors code. Colours defined in qdos.h

---

int sd_setst(long chan, timeout_t timeout, int colour)

Routine to set strip colour. Colour value (0-7) dependent on mode. Returns QDOS error code.

---

int sd_setsz(chanid_t channel, timeout_t timeout,
short c_width, short c_height)

Routine to set character width and height in a window. Possible widths are:

0 = 6 pixels wide,

1 = 8 pixels wide,

2 = 12 pixels wide,

3 = 16 pixels wide

Possible height are:

0 = 10 pixels high,

1 = 20 pixels high.

Returns QDOS error code.

---

int sd_setul(chanid_t chan, timeout_t timeout, int onoff)

Routine to set underline mode for characters on or off.

Returns QDOS error code.

---

int sd_tab(chanid_t channel, timeout_t timeout, int pos)

Routine to move to a column position (pos) on a line. Returns QDOS error code.

---

int sd_wdef(chanid_t channel, timeout_t timeout,
colour_t b_colour, short b_width,
QLRECT_t *rect)

Routine to redefine the poistion and shape of a window. The old window contents are not moved or modified, but the cursor is positioned at the top left hand corner of the new window. The values for border colour and border width are passed explicitly as parametsr. The new position and size for the window are passed as a pointer to a QLRECT_t structure whose members define the origin, height and width.

Returns QDOS error code.

---

int sms_fthg (char * thing_name, jobid_t jobid, long * d2,
long d3, char * a1, char **a2)

Free the named 'thing'. Available as standard with SMS systems, and on QDOS compatible systems with THING support code loaded. Returns the QDOS error code. The parameters d2, d3, a1 and a2 are used to pass extra parameters as defined in the definition of the 'thing' that is being freed. Note also that the d2 and a2 parameters are pointers to these values as new values can be passed back from the 'thing' being freed. The d3 and a1 parameters are not changed, so pointers are not used for these parameters.

---

int sms_lthg (THING_LINKAGE * thing_linkage)

Routine to link in a new Thing. Available as standard with SMS, and on QDOS compatible systems with THING support code loaded. The structure THING_LINKAGE is defined if you include the qdos.h or sms.h header files.

---

int sms_nthg (char * thing_name, THING_LINKAGE **next_thing)

Routine to find next Thing. Available as standard with SMS, and on QDOS compatible systems with THING support code loaded. The 'thing_name' parameter is a C style NULL terminated string. The 'next_thing' parameter is used to return the Thing Linkage block for the next Thing, or 0 if no further Thing exists. The THING_LINKAGE structure is defined in the sms.h header file. Returns SMS error code.

---

int sms_nthu (char *name, THING_LINKAGE ** thing_linkage,
jobid_t * owner_job)

SMS routine to get the owner of a job, and the next linkage block. If the pointer pointed to by thing_linkage is 0, then this the value returned in 'owner_job' is undefined, and this routine functions like the **sms_nthg()** routine.

defined in sms.h

---

int sms_rthg (char * thing_name)

SMS routine to remove a Thing if it is not in use. The 'thing_name' parameter is a C style (NULL terminated) string.

defined in sms.h

---

char * sms_uthg (char * thing_name, jobid_t job_id,
timeout_t timeout, long *d2, char *a2,
long *version, THING_LINKAGE **linkage)

SMS routine to use a Thing. The name is passed in C (NULL terminated) format. The version is returned in the 'version' parameter. The additional values passed/returned in the 'd2' and passed in the 'a2' parameters are dependent upon the definition of the THING being used. The 'linkage' parameter is used to get back the Thing linkage address on a successful call. If an error occurs, then the error code (which is negative) is returned. If successful, the address of the Thing is returned, and a pointer to its linkage in the 'linkage' parameter. The THING_LINKAGE structure is defined in the sms.h header file.

Defined in sms.h

---

int sms_zthg (char * thing_name)

Zap a thing. The name is supplied in C (NULL terminated) format. Returns SMS Error code.

Defined in sms.h

---

chanid_t ut_con(WINDOWDEF_t * wdef)

Simplified routine to open a console window The WINDOWDEF_t structure is defined in sys/qlib.h. Returns QDOS channel id on success, and QDOS error code (which is negative) on failure.

---

int ut_cstr (const QLSTR_t * string1,
const QLSTR_t * string2, int mode)

Compare two QDOS strings. The QLSTR_t strucutre is defined in sys/qlib_h. The type of comparison is determined by mode as follows:

0 Compare on a character by character basis. Case is significant

1 As type 0, but ignore case

2 Embedded numbers are converted to binary before comparison. Text characters are case significant.

3 As type 2, but case is ignored.

The order of comparison uses the QDOS defined collating sequence (which is not the same as the ASCII values of the characters). The value returned is 0 if the strings match, -1 if 'string1' is less than 'string2', and +1 if 'string1' is greater than 'string2'.

---

void ut_err(int qdoserror, chanid_t channel)

Write the message corresponding to the error code to the specified channel.

---

void ut_err0 (int qdoserror)

Write the message corresponding to the QDOS error code to channel 0.

---

void ut_link (char *previous_item, char * nextitem)

Link an item into a linked list.

---

int ut_mint(chanid_t channel, int value)

Convert a value to ASCII and send it to the specified channel. Returns QDOS error code (if any).

---

int ut_mtext(chanid_t, QLSTR * message)

Send a message to a specified channel. Returns QDOS error code (if any).

---

chanid_t ut_scr (WINDOWDEF_t * windef)

Simplified routine to open a screen window. The WINDOWDEF_t structure is defined in sys/qlib.h. Returns channel on success, QDOS errror code (which is negative) on failure.

---

void ut_unlnk (char *previous_item, char * old_item)

Unlink an item from a linked list.

---

chanid_t ut_window (char *name, char *details)

Simplified routine to open a window. the 'name' parameter is a C type string that specifies the type and dimensions. The details parameter specifies the border details and the paper/ink colours. Returns the QDOS channel id on success and a QDOS error code (which is negative) on failure.

MANIFEST CONSTANTS

There following manifest constants are defined in QODS.H for the error codes returned by QDOS.

Constant	Meaning
ERR_OK	NO error occured
ERR_BL	Bad line in BASIC
ERR_BN	Bad device name
ERR_BO	Buffer overflow
ERR_BP	Bad parameter
ERR_DF	Drive full
ERR_EF	End of file
ERR_EX	File already exists
ERR_FE	File error
ERR_FF	Format failed
ERR_IU	File or device in use
ERR_NC	Operation not complete
ERR_NF	File or device not found
ERR_NI	Not implemented
ERR_NJ	Not a valid job
ERR_NO	Channel not open
ERR_OM	Out of memory
ERR_OR	Out of range
ERR_OV	Arithmetic overflow
ERR_RO	Read only
ERR_RW	Read or Write Failed
ERR_TE	Transmission error
ERR_XP	Error in expression

CHANGE HISTORY

20 Jun 93	Added descriptions for the Queue Handling routines io_qeof(), io_qin(), io_qout(), io_qset(), io_qtest(), io_serq(), io_serio().
10 Jul 93	Description of the majority of the trap calls amended to remove the statement that they set the _oserr global variable (where this is no longer true).
08 Sep 93	Added c_extop() call (based on a contribution by PROGS of Belgium).
31 Dec 93	Documented the iop_outl() call.
24 Jan 94	Reworked this document to only include the direct calls to QDOS. Direct calls under SMS names are now documented in LIBSMS_DOC, and all more generic alls on LIBC68_DOC. Added the names of the standard QDOS error codes as manifest constants.

routines.

10 Jun 94 The cross-reference list of the routines by function removed from this document. All such lists are now consolidated into the LIBINDEX_DOC file.

Pointer Environment Library

INTRODUCTION

The **libqptr** library is designed to allow you to write programs that exploit the Pointer Environment. The Pointer Environment is built into SMS2 systems, but need to be explicitly loaded for systems running standard QDOS or SMSQ.

You should bear in mind that the Pointer Environment is very specific to the QDOS, SMSQ and SMS2 fmaily of operating systems. If you use these facilities it will not be easy (or sometime not even possible) to port such programs to other operating environments. You should bear this fact in mind when you decide to use the routines in the **libqptr** library.

You do not ever need to tell the linker explicitly that you want to include routines from the **libqptr** library. The routines defined as being in the **libqptr** library are imbedded in the LIBC_A library which is automatically included at the end of the link by the LD linker. You should always, however have the statement

#include <qptr.h>

in any program or module that makes use of the routines defined as being in this library. If you do not you will get error messages from the linker stating that the LIBQPTR routines are undefined.

TYPEDEF'ed STRUCTURES

To help you to produce readable code, all the structures used in the LIBQPTR_A routines have been typedef'ed. The names of the typedef are always constructed by adding '_t' to the structure name.

This means that instead of writing something like

struct WM_wdef

you can use

WM_wdef_t

which is slightly more readable, and also helps the compiler do stricter type checking.

REFERENCE MATERIAL

The reference books listed below were used in preparing the material for inclusion in this library:

"QPTR Pointer Environment" manual sold by Jochen Merz

LIBRARY CONTENTS

The routines in this library are split into the following sections:

- Button Frame Utility Functions,
- Window Manager Utility Functions (C68 compatible),
- Window Manager Wrappers and Internal Routines (not callable from C68),
- Pointer Interface Trap Wrappers.

BUTTON FRAME UTILITY FUNCTIONS

---

int bt_frame (chanid_t, WM_swdef_t *sw)

Using size and attributes from the sub-window definition sw, bt_frame allocates a space in the button frame for the channel, sets the origin in sw, sets border/paper/strip and, if specified, clears the window.

---

int bt_free (void)

Frees the button frame allocation.

---

int bt_prpos (WM_wwork_t *)

As wm_prpos but positions primary window in button frame. If successful, the shadow width is set to zero. Returns QDOS/SMS error code.

POINTER INTERFACE CALLS

These routines allow C programs to call the Pointer Interface system calls.

---

int iop_flim (chanid_t, timeout_t, WM_wsiz_t * limits)

Find window limits. Values returned via the 'limits' parameter. Returns standard QDOS/SMS error codes.

---

int iop_lblb (chanid_t, timeout_t, short xs, short ys,
short xe, short ye, WM_blob_t *, WM_pattern_t *)

Draw a line of blobs from xs,ys to xe,ye. Returns standard QDOS/SMS error codes.

---

int iop_outl (chanid_t, timeout_t, short shadx, short shady,
short keep(0/1), WM_wsiz_t *)

Set outline. Keep = 1 to keep contents. Returns standard QDOS/SMS error codes,

---

int iop_pick (chanid_t, timeout_t, jobid_t job_ID)

Pick windows for a Job to the top. Returns standard QDOS/SMS error codes.

---

void * iop_pinf (chanid_t, timeout_t, long *version)

Get Window Manager vector and version information. On success returns the address of the Window Manager vector. On failure returns the QDOS/SMS error code (which is negative).

N.B. Prior to Release 3 of the QPTR library, a value of 0 was returned on error. This means that code that used that version might need slight modification in its use of this call.

---

int iop_rptr (chanid_t, timeout_t, short *x, short *y,
short termination_vector, WM_prec_t *)

Reads pointer and suspend until termination conditions (as specified in the termination vector parameter) or timeout occurs. Returns QDOS/SMS error code.

N.B. Prior to Release 3 of the QPTR library, the x and y parameters were treated as though they pointed to 'int' rather than 'short' values as the specification said. This means that code that used that version might need slight modification in its use of this call.

---

int iop_rpxl (chanid_t, timeout_t, short *x, short *y,
short scan, short *pixel)

Read pixel. Returns QDOS/SMS error code.

N.B. Prior to Release 3 of the QPTR library, the x and y parameters were treated as though they pointed to 'int' rather than 'short' values as the specification said. This means that code that used that version might need slight modification in its use of this call.

---

int iop_rspw (chanid_t, timeout_t, WM_wsiz_t *save,
short xorg, short yorg,
int keepflag, void *save_area)

Restore partial window. Returns QDOS/SMS error code.

---

void * iop_slnk (chanid_t, timeout_t, void * values,
short start, short count)

Set pointer linkage. On success returns the base address of the linkage. On failure returns a QDOS/SMS error code (which is a negative value).

---

int iop_spry (chanid_t, timeout_t, short x, short y,
WM_blob_t *, WM_pattern_t *, long num_pixels)

Spray pixels of pattern within blob. Returns QDOS/SMS error code.

---

int iop_sptr (chanid_t, timeout_t, short *x, short *y,
char origin_key)

Set pointer absolute (origin=0), relative to hit area (origin=-1). Sets the absolute pointer position in x,y.

---

int iop_svpw (chanid_t, timeout_t, WM_wsiz_t *,
short xorg, short yorg, short xsize,
short ysize, void **save_area)

Save partial window. If xsize and ysize are zero, then the area should already exist. If they are non-zero then a new save area is set up and its address stored at the location specified by the 'save_area' parameter.

Returns QDOS/SMS error code.

---

int iop_swdf (chanid_t, timeout_t, long *wdef_list)

int iop_swdl (chanid_t, timeout_t, long *wdef_list)

Sets window definition list. Returns QDOS/SMS error code.

---

int iop_wblb (chanid_t, timeout_t, short x, short y,
WM_blob_t *, WM_pattern_t *)

Write blob. Returns QDOS/SMS error code.

---

int iop_wrst (chanid_t, timeout_t, void *save, char keep)

Restore window. If save==NULL, then do it form internal area, otherwise use area supplied by user. If keep!=0 then keep save area. Returns QDOS/SMS error code.

---

int iop_wsav (chanid_t, timeout_t, void *save, long length)

Save window area. If save==NULL and length==0, then the area is allocated internally. If not, area supplied by user is used. Returns QDOS/SMS error code.

---

int iop_wspt (chanid_t, timeout_t, short x, short y,
WM_sprite_t *)

Write sprite. Returns QDOS/SMS error code.

WINDOW MANAGER FUNCTIONS (C68 compatible)

These are C equivalents to the standard Window Manager calls available to assembler programmers. More details can be found in the QPTR manual.

---

int wm_chwin (WM_wwork_t *, short *dx, short *dy)

Change window position (automatic) or size (returns the dx,dy of the pointer). Returns QDOS/SMS error code on failure, 0 or a positive event number if successful.

---

int wm_clbdr (WM_wwork_t *)

If there is a current item, it is cleared: useful before re-drawing menus. Returns QDOS/SMS error code.

N.B. This is a C68 extension to the standard Window Manager set of vectors.

---

int wm_cluns (WM_wwork_t *)

Close channel and unset window. (Actually a call to wm_unset then the channel is closed. Use it to get rid of pull-down windows. Returns QDOS/SMS error code.

N.B. This is a C68 extension to the standard Window Manager set of vectors.

---

int wm_drbdr (WM_wwork_t *)

Draws a border using the current item definition in WM_wstat. Returns QDOS/SMS error code.

---

int wm_ename (chanid_t, QD_text_t * name)

Edit name (QDOS string): writes out current name, puts cursor at end. Returns QDOS/SMS error code.

C.f. wm_rname.

---

int wm_erstr (long error_code, QD_text_t * reply_string)

Converts the error code to a QDOS string. Returns the QDOS/SMS error code.

---

void * wm_findv (chanid_t channel)

Check that the Window Manager has been loaded, and if so get the Window Manager Vector: Returns the vector or NULL if not found. It is not necesary to use this call if you have already used the iop_pinf() call to check for the presence of the Window Manager. This routine also stores the value of the Window Manager vector internally for use by the other wm_xxxx calls so that the user need not store the value.

N.B. This is a C68 extension to the standard Windows Manager set of vectors.

---

short wm_fsize (short *xsize, short *ysize, WM_wdef_t *)

Given a target size and a window definition, this routine returns the appropriate layout number and sets the size to the actual size. Returns layout size (or QDOS/SMS error code if Window Manager Vector not known).

C.f. wm_setup.

C.f wm_setup.

---

int wm_idraw (WM_wwork_t *, long bits)

Re-draws any of information windows 0-31. For each window required to be drawn, the corresponding bit in bits should be set. Returns QDOS/SMS error code.

---

int wm_index (WM_wwork_t *, WM_swdef_t *)

Draws the index (not implemented), pan and scroll bars for an application sub-window.

---

int wm_ldraw (WM_wwork_t *, char select)

Loose menu Item Drawing. Returns QDOS/SMS error codes.

---

int wm_mdraw (WM_wwork_t *, WM_swdef_t *, int select)

Draws all menu items (select =0) or those items with change bit set in status area (select<>0).

---

int wm_mhit (WM_wwork_t *, WM_appw_t *, short x, short y,
short key, short event)

C68 compatible wrapper for wm.mhit. Can be called from application sub-window hit routine.

C.f. wm__mhit.

---

short wm_msect (WM_wwork_t *, WM_appw_t *, short xpos,
short ypos, short key, short event, WM_mctrl_t *)

Called from an application sub-window hit routine, wm_msect determines the section of a menu and whether a pan or scroll event has occurred. The general information is returned in the structure WM_mctrl. If there has been an pan/scroll event, this is returned (+ve) otherwise wm_msect returns 0 or a QDOS/SMS error code.

---

int wm_pansc (WM_wwork_t *, WM_appw_t *, WM_mctrl_t *)

If wm_msect returns a pan or scroll event: this routine can handle it.

---

int wm_prpos (WM_wwork_t *, short xpos, short ypos)

Position Primary Window. Returns QDOS/SMS error code.

---

int wm_pulld (WM_wwork_t *, short xpos, short ypos)

Pull down a secondary window. Retruns QDOS/SMS error code.

---

int wm_rname (chanid_t, QD_text_t *)

Read name (QDOS string): writes out current name, puts cursor at start. Typing any printable character erases name. Returns QDOS/SMS error code.

C.f. wm_ename.

---

int wm_rptr (WM_wwork_t *)

Returns QDOS/SMS error code.

---

int wm_setup (chanid_t, short xsize, short ysize, WM_wdef_t *,
WM_wstat_t *, WM_wwork_t **, long alloc)

If the alloc size is non-zero, then a new Working Defintion area of this size will be allocated on the common heap. If it is zero, then it is assumed that the area is already allocated.

Returns QDOS/SMS error code (if Window Manager Vector cannot be located).

---

int wm_smenu (short xscale, short yscale, WM_wstat_t *,
WM_wdef_t **, WM_wwork_t **)

Setup standard sub-window menu. Returns QDOS/SMS error code if unable to find Window Manager.

---

int wm_stiob (WM_wwork_t *, void *object,
short window nr, short object number)

Set information object. Returns QDOS/SMS error code.

---

int wm stlob (WM wwork t *, void *; short item number)

Set loose object. Returns QDOS/SMS error code.

---

chanid_t wm_swapp (WM_wwork_t *, short window nr, long ink)

Set window to application window. Returns channel ID or QDOS/SMS error code.

---

chanid_t wm_swdef (WM_wwork_t *, WM_appw_t *, chanid_t channel)

Set channel to application sub-window. Does not set colours. Returns Channel ID or QDOS/SMS error code.

---

chanid_t wm_swinf (WM_wwork_t *, short window nr, long ink)

Set window to information window. Returns channel ID or QDOS/SMS error code.

---

chanid_t wm_swlit (WM_wwork_t *, short window nr, long status)

Set window to loose item. Returns channel ID or QDOS/SMS error code.

---

chanid_t wm_swsec (WM_wwork_t *, WM_appw_t *, short xsection, short ysection, long ink)

Set window to application sub-window section. Returns channel ID or QDOS/SMS error code.

---

int wm_unset (WM_wwork_t *)

Unset window: obligatory before scrumpling the working definition. Also used to remove pull-down windows. Returns QDOS/SMS error code.

C.f. wm_cluns.

---

int wm_upbar (WM_wwork_t *, WM_swdef_t *, short xsection, short ysection)

Update a section of the pan/scroll bar.

---

int wm_wdraw (WM_wwork_t *)

Draw window: after wm_prpos or wm_pulld. Returns QDOS/SMS error code.

---

int wm_wrset (WM_wwork_t *)

Reset window definition. Returns QDOS/SMS error code.

Window Manager Routines Referenced From Working Definition

---

int wm_smenu (...)

referenced from wda_setr (assembly language)

---

int wm__mhit (...)

referenced from WM_appw.hit

---

int wm__pnsc (...)

referenced from WM_appw.ctrl

Window Manager Action (etc) Routine Wrappers

These wrappers allow C68 functions to be called from the Window Manager via the WM_action structure.

---

wm_actli(...)

referenced from WM_litm.pact

---

wm_actme(...)

referenced from WM_mobj.pact

---

wm_drwaw(...)

referenced from WM_appw.draw

---

wm_hitaw(...)

referenced from WM_appw.hit

---

wm_ctlaw(...)

referenced from WM_appw.ctrl

STANDARD SPRITES

The following pre-defined sprites that are commonly used in Pointer Environment programs are included in this library.

Any further contributions that could be added to this standard sprite list would be welcomed.

---

struct WM_sprite wm_sprite_arrow

Arrow symbol

---

struct WM_sprite wm_sprite_cf1

CTRL-F1 key symbol

---

struct WM_sprite wm_sprite_cf2

CTRL-F2 key symbol

---

struct WM_sprite wm_sprite_cf3

CTRL-F3 key symbol

---

struct WM_sprite wm_sprite_cf4

CTRL-F4 key symbol

---

struct WM_sprite wm_sprite_f1

F1 key symbol

---

struct WM_sprite wm_sprite_f2

F2 key symbol

---

struct WM_sprite wm_sprite_f3

F3 key symbol

---

struct WM_sprite wm_sprite_f4

F4 key symbol

---

struct WM_sprite wm_sprite_f5

F5 key symbol

---

struct WM_sprite wm_sprite_f6

F6 key symbol

---

struct WM_sprite wm_sprite_f7

F7 key symbol

---

struct WM_sprite wm_sprite_f8

F8 key symbol

---

struct WM_sprite wm_sprite_f9

F9 key symbol

---

struct WM_sprite wm_sprite_f10

F10 key symbol

---

struct WM_sprite wm_sprite_hand

Hand symbol

---

struct WM_sprite wm_sprite_insg

---

struct WM_sprite wm_sprite_insl

---

struct WM_sprite wm_sprite_left

---

struct WM_sprite wm_sprite_move

Move symbol. Used to indicate item used a window.

---

struct WM_sprite wm_sprite_null

---

struct WM_sprite wm_sprite_size

Size symbol. Used to indicate menu item that is used to re-size a window.

---

struct WM_sprite wm_sprite_sleep

Sleep symbol. Used to indicate menu item for putting a program to sleep.

---

struct WM_sprite wm_sprite_wake

Wake symbol. Used to indicate a menu item for waking a program.

---

struct WM_sprite wm_sprite_zero

This is really just a blank background. It is used as the pattern mask for many of the sprites.

CHANGE HISTORY

The following is a brief summary of the significant changes made to this document. It is intended to help those who are upgrading from previous releases to determine what (if anything) has changed in this document.

30 Oct 93	DJW - Extensive changes as part of making the QPTR library usable with C68 Release 4.
02 Nov 93	DJW - Added list of sprites that are included in this library.
13 Aug 94	DJW - Changed the definition of the iop_rspw() routine to make the last parameter only 'void *' (it was 'void **').
03 Apr 95	DJW - Changed all function definitions reflect fact that all structures are now 'typedef'ed. Also 'char *' parameters changed to more generic 'void *' format.

SMS/SMSQ/SMSQ-E System Interface

This section of the C68 library documentation covers those routines in the C68 standard library that provide access to the SMS operating system interfaces.

All of the calls in this part of the library map directly onto the SMS System Calls available to Assembler (machine code) programmers. It is therefore useful to have access to documentation covering the Assembler level interface to SMS if you want more details on how many of these calls work.

You do not ever need to tell the linker explicitly that you want to include routines defined in this document. These routines are imbedded in the LIBC_A library which is included automatically the LD linker. You must always, however have the statement

#include <sms.h>

in any program or module that makes use of the routines in this library. If any additional header is required as well, this will be mentioned in the description of the routine.

It is worth noting that most of the calls defined here also work on the QDOS family of operating systems. However in that case they traditionally have alternative names. If you wish to find the functions listed and described under their QDOS names, then refer to the LIBQDOS_DOC file. If any call does not work in both environments this is mentioned in the description of the function.

REFERENCE MATERIAL

The reference books listed below were used in preparing material for inclusion in this library:

QDOS/SMS Reference Manual

as published by Jochen Merz

---

int c_extop (chanid_t channel, timeout_t timeout,
int (*func), int number_of_params, ...)

Allow a routine to be called to do an extended operation on a QDOS channel. The parameters are passed in a way that is compatible with this routine being written in C (c.f. sd_extop()/iow_xtop() for assembler only routines).

The C routine will be called in supervisor mode, with the parameters

The C routine will be called in supervisor mode, with the parameters specified by ... above passed to it on the stack. Each parameter is assumed to be no larger than 4 bytes in size (i.e. no structures are to be passed on the stack). Note also that due to a bug in QDOS, it seems to hang if the routine does not return zero in D0. Therefore, if it is desired to pass an error code back to the application program it must be done indirectly via one of the parameters.

---

void cv_fpdec (char * target, char * value)

Convert a SMS floating point value into a decimal character ASCII string.

---

void cv_ibbin (char * target, char * value)

Convert a byte into a 8 character ASCII string of binary.

---

void cv_ibhex (char * target, char * value)

Convert a byte into a 2 character ASCII hex string.

---

char *cv_ildat(char *asciidate, time_t qldate)

Converts a date from internal SMS format into an ASCII string in the format "YYYY mmm dd hh:mm:ss". The asciidate parameter must point to a buffer of at least 25 characters in length to hold the return data. The buffer returned is in SMS string format - which is a 2 byte length field, followed by the data (NULL terminated for convenience to C programmers). The return value is the address of the start of the text.

---

char *void cv_ilday(char *asciiday, time_t qldate)

Returns the 3 character day of the week given a date in SMS internal format. The asciidate parameter must point to a buffer of at least 7 characters in length to hold the return data. The buffer returned is in SMS string format - which is a 2 byte length field, followed by the data (NULL terminated for convenience to C programmers). The return value is the address of the start of the text.

---

void cv_ilbin (char * target, long * value)

Convert a long integer into a 32 character ASCII string of binary.

---

void cv_ilhex (char * target, long * value)

Convert a long integer into a 8 character ASCII hex string.

---

void cv_iwbin (char * target, short * value)

Convert a short integer (word) into a 16 character ASCII string of binary.

---

void cv_iwdec (char * target, short * value)

Convert a short integer into a decimal ASCII string.

---

void cv_iwhex (char * target, short * value)

Convert a short integer (word) into a 4 character ASCII hex string.

---

int ioa_cnam (chanid_t channel, char * buffer,
short buffer_length)

Get name of a channel. The buffer will be filled in with the channel name. It must be large enough to contain the channel name, a terminating NULL byte and one additional byte.

NOTE. Only available on SMSQ and SMSQ/E based systems

---

int ioa_sown (chanid_t channel, jobid_t new_owner)

Set owner of a channel.

NOTE. Only available on SMSQ and SMSQ/E based systems

---

int iob_elin (chanid_t channel, timeout_t timeout,
char **cptr, short bufsize,
short current_offset, short *current_linelen);

Edited line read call. Returns SMS error code.

---

int iob_fbyt(chanid_t channel, timeout_t timeout,
char *char_pointer)

Read 1 byte. Returns SMS error code.

---

int iob_flin(chanid_t channel, timeout_t timeout,
void *buf, short length)

Read a linefeed terminated string of bytes. Returns length read on success, SMS error code (which is negative) on failure.

---

int iob_fmul(chanid_t channel, timeout_t timeout,
void *buf, short length)

Fetch a string of bytes. Returns length read on success, or SMS error code (which is negative) on failure. The amount read can be less than the amount requested. This would normally caused by an end-of-file or timeout condition occuring during the read.

---

int iob_sbyt(chanid_t channel, timeout_t timeout,
unsigned char ch)

Output char 'ch' to channel. Returns SMS error code.

---

int iob_smul(chanid_t chid, timeout_t, void *buf, short len)

Write a string of bytes. Returns length written on success, and a SMS error code (which is negative) on failure. The amount written can be less than the amount requested. This would normally be caused by a timeout condition occuring during the write.

---

int iob_suml (chanid_t chid, timeout_t, void *buf, short len)

Write a string of untranslated bytes. This is very similar to the **iob_smul()** call except that the settings of translation calls is ignored as well as any character translation implied in the device open call (e.g. SERd, SERt, PARd, PARt). This is a safe way of sending graphics data or control codes to the device as they will never be translated to any other characters. Values returned are the same as for the **iob_smul()** call.

This call is only supported on SMSQ and SMSQ/E based systems. Other systems will return an error code for an unimplemented trap call.

---

int iob_test(chanid_t chan, timeout_t timeout)

Test for any pending input on a channel, returns 0 if data is to be read, else -1 (not complete).

---

int iof_date(chanid_t chan, timeout_t timeout, int type,
long * sr_date)

type = 0 Access update date of file,

= 2 Access backup date.

*sr_date = -1 Read requested date (returned from call in *sr_date)

= 0 Set requested date to current date.

else Set requested date to date given in *sr_date.

Read/Set update or backup dates. Available on Miracle Systems hard disk, ST/QL sytems and SMS systems. The date set/read is returned in *sr_date. Returns SMS error code.

---

int iof_flsh(chanid_t channel, timeout_t timeout)

Flush all buffers on a file. Returns SMS error codes.

---

long iof_load(chanid_t channel, char * buf,
unsigned long len)

Load a complete file. Returns length loaded on sucess, SMS error code (which is negative) on error.

---

int iof_minf(chanid_t chan, timeout_t timeout,char * medname,
short * unused_secs, short * goodsecs)

Get media information. Returns 10 character name of media (N.B. not NULL terminated), number of unused_sectors, and number of good sectors. Returns SMS error code.

---

int iof_mkdr(chanid_t channel)

Make the file specified by the SMS channel into a directory. Requires support for Level 2 filing system (e.g. Miracle hard Disk, ST/QL or SMS systems). Returns SMS error code.

---

long iof_pos(chanid_t chan, long pos, int mode)

long iof_pos(chanid_t chan, long pos, int mode)

SMS equivalent to C seek() routine to seek to a point in a file (no timeout as it's always -1). mode can have the following values:

0 absolute

1 relative to current position

2 relative to EOF.

Returns new position on sucess, and SMS error code (which is negative) on failure.

---

long iof_posa(chanid_t chan, timeout_t timeout,
unsigned long * pos)

Seek to an absolute point in a file. The new file position is returned via the 'pos' poarameter. Returns SMS error code.

---

long iof_posr(chanid_t chan, timeout_t timeout, long * pos)

Seek to a point in a file relative to the current position. The new file position is returned via the 'pos' poarameter. Returns SMS error code.

---

int iof_rhdr(chanid_t chan, timeout_t timeout,
void * buf, short buflen)

Read a file header. Returns length read on success, SMS error code (which is negative) on failure.

---

int iof_rnam(char * old, char * new)

Rename a file. Uses C strings. Very basic QL systems (without Toolkit 2) may not support this call, but any other type of system can be expected to support it. Returns SMS error code.

---

int iof_save(chanid_t channel, char * buf,
unsigned long len)

Save a complete file to a channel. Returns length saved on success, SMS error code (which is negative) on failure.

---

int iof_shdr(chanid_t chan, timeout_t timeout,
void * buf, short buflen)

Set a file header. Returns length read on success, SMS error code (which is negative) on failure.

---

int iof_trunc(chanid_t channel, timeout_t timeout)

Truncate a file at the current byte position. This call may not be available on very basic QL systems (unless Toolkit 2 present) but all other types of system can be expected to support it. Returns SMS error code.

---

int iof_vers(chanid_t channel, timeout_t timeout, long *key)

Set/Read a file version number. Only available on systems that support version 2 (or better) filing systems (such as Miracle hard disk, ST/QL and SMS systems). The action is defined as follows:

*key = -1 Return version number in *key.

= 0 Keep old version number when file closed (return it on *key)

= +ve and < 65536 Set version number to given number.

Returns SMS error code.

---

int iof_xinf(chanid_t channel, timeout_t timeout,
struct ext_mdinf * fsinf)

Get extended file system info. Only available on systems that support version2 filing system (such as Miracle hard disk, ST/QL and SMS systems). Requested data is returned in struct ext_mdinf (defined in qdos.h) on success. Returns SMS error code.

---

int iog_arc(chanid_t channel, timeout_t timeout,
double x_start, double y_start,
double x_end, double y_end, double angle)

Draw an arc using graphics coordinates. sd_arc uses C double precision floating point coordinates (cf. sd_iarc). Returns SMS error code.

---

int iog_arc_i(chanid_t channel, timeout_t timeout,
double x_start, double y_start,
double x_end, double y_end, double angle)

double x_end, double y_end, double angle,

Draw an arc using graphics coordinates. iog_arc_i takes integer coordinates (c.f. iog_arc) Returns SMS error code.

---

int iog_dot(chanid_t channel, timeout_t timeout,
double x, double y)

Plot a point using graphics coordinates. iog_dot takes C double precision floating point coordinates (cf. iog_dot). Returns SMS error code.

---

int iog_dot_i(chanid_t channel, timeout_t timeout,
int x, int y)

Plot a point using graphics coordinates. iog_dot_i takes integer coordinates (cf. iog_dot). Returns SMS error code.

---

int iog_elip (chanid_t channel, timeout_t timeout,
double x_centre, double y_centre,
double eccentricity, double radius,
double angle_of_rotation)

Draw a circle or ellipse using graphics coordinates. iog_elip uses C double precision floating point coordinates (cf. iog_elip_i). Returns SMS error code.

---

int iog_elip_i(chanid_t channel, timeout_t timeout,
int x_centre, int y_centre, int eccentricity, int radius, int angle_of_rotation)

Draw a circle or ellipse using graphics coordinates. iog_elip_i uses integer coordinates (cf. iog_elip). Returns SMS error code.

---

int iog_fill(chanid_t channel, timeout_t timeout, int onoff)

Set flood fill mode on or off. Returns SMS error code.

---

int iog_line(chanid_t channel, timeout_t timeout,
double x_start, double y_start,
double x_end, double y_end)

Draw a line with graphics coordinates. iog_line uses C double precision floating point coordinates (cf. iog_line_i). Returns SMS error code.

---

int iog_line_i(chanid_t channel, timeout_t timeout,
int x_start,int y_start, int x_end, int y_end)

Draw a line with graphics coordinates. iog_line_i takes integer coordinates (cf. iog_line). Returns SMS error code.

---

int iog_scal (chanid_t channel, timeout_t timeout,
double scale,double x_origin, double y_origin)

Change a windows graphics origin and scale. iog_scal uses C double precision floating point coordinates (cf. iog_scal_i). Returns SMS error code.

---

int iog_scal_i chanid_t channel, timeout_t timeout,
int scale, int x_origin, int y_origin)

Change a windows graphics origin and scale. iog_scal_i uses integer coordinates (cf. iog_scal). Returns SMS error code.

---

int iog_sgcr(chanid_t channel, timeout_t timeout,
double vert_offset, double horiz_offset,
double x_pos, double y_pos)

Set the graphics text cursor. iog_sgcr uses C double precision floating point coordinates (cf. iog_sgcr_i). Returns SMS error code.

---

int iog_sgcr_i(chanid_t channel, timeout_t timeout,
int vert_offset, int horiz_offset,
int x_pos, int y_pos)

Set the graphics text cursor. iog_sgcr_i uses integer coordinates (cf. iog_sgcr). Returns SMS error code.

---

int iop_outl (chanid_t channel, timeout_t timeout,
short, short, short, void *)

This is the call that sets the outline window for a Pointer Environment. It

This is the call that sets the outline window for a Pointer Environment. It is included in this library as it is the one call that need to be issued to make a program that is not otherwise aware of the pointer environment function correctly in that environment.

For more details refer to the LIBQPTR_DOC file provided as part of the QPTR library.

Note that the default console initialisation routines supplied with C68 will automatically issue a call to set the window outline to the size as defined in the '_condetails' global variable (see LIBC68_DOC for more details).

---

int ioq_gbyt (char * queue_pointer, char * next_byte)

Remove a byte from a queue. Returns the SMS error code (if any).

---

int ioq_pbyt (char * queue_pointer, int byte_to_insert)

Insert a byte in a queue. Returns the SMS error code (if any).

---

int ioq_seof(char * queue_pointer)

Insert an EOF (end-of-file) marker into a queue. Returns SMS error code (if any).

---

void ioq_setq(char * queue_pointer, long queue_length)

Set up a queue.

---

int ioq_test(char * queue_pointer, char * next_byte,
long * free_space)

Test the status of a queue. The variables whose addresses are passed as parameters are updated to the free space in the queue, and (if there is data in the queue) the value of the next byte is returned (although the byte is not removed from the queue). The SMS error code is returned.

---

int iou_ssio(chanid_t channel_id, timeout_t timeout,
int routine_number, long * D1, long * D2,
char ** A1, char * routine_array[4])

General serial IO handling routine. This routine is used when the **iou_ssq()** routine is not sufficient. The values passed as the parameters 'D1', 'D2' and 'A1' are pointers to the values to be put into the registers D1, D2 and A1 respectively. These values may be changed by this routine. The 'routine_array' is an array of at least 4 elements, the first three of which contain the addresses of the routines for testing pending input, fetching a byte and sending a byte. The fourth element will be used as workspace, and thus corrupted by this call.

---

int iou_ssq (chanid_t channel_id, timetout_t timeout,
int routine_number, long * D1, long * D2,
char ** A1)

Serial IO Direct Queue handling routine. The values passed as the parameters 'D1', 'D2' and 'A1' are pointers to the values to be put into the registers D1, D2 and A1 respectively. These values may be changed by this routine.

---

int iow_blokl(chanid_t channel, timeout_t timeout,
colour_t colour, QLRECT_t * rect)

Plot a rectangular block of a certain colour. Can be used to draw very fast horizontal and vertical lines. Returns SMS error code.

---

int iow_chrq(chanid_t channel, timeout_t, QLRECT_t * rect)

Read a window size in characters. On success 'rect' is set to details of answer. Returns SMS error code.

---

int iow_clra(chanid_t channel, timeout_t timeout)

Clear entire window.

Returns SMS error code

---

int iow_clrb(chanid_t channel, timeout_t timeout)

Clear area of window below cursor line.

Returns SMS error code.

---

int iow_clrl(chanid_t channel, timeout_t timeout)

int iow_clrl(chanid_t channel, timeout_t timeout)

Clear all of cursor line. Returns SMS error code.

---

int iow_clrr(chanid_t channel, timeout_t timeout)

Clear cursor line, to right of cursor position (including cursor). Returns SMS error code.

---

int iow_clrt(chanid_t channel, timeout_t timeout)

Clear area of window above cursor line. Returns SMS error code.

---

int iow_dcur(long chan, timeout_t timeout)

Disable cursor on screen channel. Returns SMS error code.

---

int iow_defb (chanid_t channel, timeout_t timeout,
unsigned char colour, short width)

Redifine a window border with new colour and width. Returns SMS error codes.

---

int iow_defw(chanid_t channel, timeout_t timeout,
colour_t b_colour, short b_width,QLRECT_t *rect)

Redefine a window size and border, given new border colour and size and new window size as a QLRECT_t structure. Returns SMS error code.

---

int iow_donl(chanid_t channel, timeout_t timeout)

Flush any pending newlines on a window channel. Returns SMS error code.

---

int iow_ecur(chanid_t channel, timeout_t timeout)

Enables cursor on screen channel. Returns SMS error code.

---

int iow_font(chanid_t channel, timeout_t timeout,
void *font1, void * font2)

Set normal and alternative character font in a window. Passed pointers to two font definitions (format as described in SMS manuals). Returns SMS error code.

---

int iow_font_def(chanid_t channel, timeout_t timeout,
void * font1, void * font2)

Set or reset the default system font. Passed pointers to two font definitions (format as described in SMS manuals). Each of the 'font1' and 'font2' parameters can also take the values of -1 to keep its current setting, or 0 to select the default font built into the system. Returns SMS error code.

NOTE. Only available on SMSQ or SMSQ/E based systems.

---

int iow_ncol(chanid_t channel, timeout_t timeout)

Move cursor right one column. Returns SMS error code.

---

int iow_newl(chanid_t channel, timeout_t timeout)

Move cursor to start of next line. Returns SMS error code.

---

int iow_nrow(chanid_t channel, timeout_t timeout)

Move cursor down one row. Returns SMS error code.

---

int iow_pana(chanid_t channel,timeout_t timeout, int ampix)

Pan window left or right. ampix < 0 means pan left, ampix > 0 means pan right. Returns SMS error code.

---

int iow_panl(chanid_t channel,timeout_t timeout, int ampix)

Pan cursor line left or right. ampix < 0 means pan left, ampix > 0 means pan right. Returns SMS error code.

---

int iow_panr(chanid_t channel,timeout_t timeout, int ampix)

Pan right of cursor line left or right (includes character at cursor position). ampix < 0 means pan left, ampix > 0 means pan right. Returns SMS errors code.

---

int iow_pcol(chanid_t channel, timeout_t timeout)

Move cursor left one column. Returns SMS error code.

---

int iow_pixq(chanid_t channel, timeout_t, QLRECT_t * rect)

Read a window size in pixels. Returns size in a QLRECT_t structure (defined in sys/qlib.h). Returns SMS error code.

---

int iow_prow(chanid_t channel, timeout_t timeout)

Move cursor up one row. Returns SMS error code.

---

int iow_rclr(chanid_t channel, timeout_t timeout,
char *collist)

Recolour a window. Done in software and very slow. colist points to eight characters containing new colours for eight possible colours. Returns SMS error code.

---

int iow_scol(chanid_t channel, timeout_t timeout, int pos)

Move to a column position (pos) on a line. Returns SMS error code.

---

int iow_scra(chanid_t channel,timeout_t timeout, int ampix)

Scroll entire window up or down. ampix < 0 means scroll down, ampix > 0 means scroll up. Returns SMS error code.

---

int iow_scrb(chanid_t channel,timeout_t timeout, int ampix)

Scroll window below cursor line up or down. ampix < 0 means scroll down, ampix > 0 means scroll up. Returns SMS error code.

---

int iow_scrt(chanid_t channel,timeout_t timeout, int ampix)

Scroll window above cursor line up or down.

ampix < 0 means scroll down, ampix > 0 means scroll up. Returns SMS error code.

---

int iow_scur(chanid_t channel, timeout_t timeout,
short x_pos, short y_pos)

Reposition the cursor to an x, y character position in a window. Returns SMS error code.

---

int iow_sflal(long chan, timeout_t timeout, int onoff)

Set flash mode on or off (only works in 8 colour mode). Returns SMS error code.

---

int iow_sink(long chan, timeout_t timeout, int colour)

Set ink colour. Colour value (0-7) dependent on mode. Returns SMS error code.

---

int iow_sova(chanid_t channel, timeout_t timeout, int mode)

Set type of drawing mode (DM_XOR, DM_OVER, DM_OR). Returns SMS error code.

---

int iow_spap(chanid_t channel,timeout_t timeout,int colour)

Set paper colour. Colour value (0-7) dependent on mode. Returns QDOS errors code. Colours defined in qdos.h

---

int iow_spix(chanid_t channel, timeout_t timeout,
short x_pos, short y_pos)

Reposition the cursor to an x, y pixel position in a window. Returns SMS error code.

---

int iow_ssiz(chanid_t channel, timeout_t timeout,
short c_width, short c_height)

Set character width and height in a window. Possible widths:

0 = 6 pixels wide,

1 = 8 pixels wide,

2 = 12 pixels wide,

3 = 16 pixels wide

Possible height are:

0 = 10 pixels high,

1 = 20 pixels high.

Returns SMS error code.

---

int iow_sstr(chanid_t chan, timeout_t timeout, int colour)

Set strip colour. Colour value (0-7) dependent on mode. Returns SMS error code.

---

int iow_sula(chanid_t chan, timeout_t timeout, int onoff)

Set underline mode for characters on or off.

Returns SMS error code.

---

int iow_xtop(chanid_t channel,timeout_t timeout,int (*rtn)(),
long paramd1, long paramd2, void *parama1)

Do extended operation on screen channel. Passed address of routine to call and parameters for d1, d2 and a1. Returns SMS error code. See also **c_extop() .**

NOTE. Due to a bug in QDOS, it appears that D0 must always be sero on exiting the rtn() function. Any error code therefore needs to be passed back indirectly via one of the other parameters.

---

char * mem_achp(long size, long *sizegot)

Allocate memory from common heap. It is passed the requested size and returns address of area allocated (or a SMS error code on failure). The area will always be allocated with the current job as the owner. If you are not interested in the true size obtained, then set 'sizegot' to NULL. Otherwise set it to the address of a variable that will be set to contain the actual size obtained (Note that even if the call succeeds this may not be the same as the size requested, as the amount requested is often rounded up by SMS. It is recommended that you use **sms_achp()** in preference to **mem_achp()** unless you are sure you know what you are doing.

WARNING

The size requested must allow for the SMS heap header, and the address returned is the start of the area allocated - not the useable area. This is in contrast to the **sms_achp()** call for which the user does not have to worry about the SMS heap header.

---

void mem_llst(char *area, char **ptr, long len)

Link an area back into a user heap area. Given area to link in, pointer to pointer to free space, and length to link in. This call is also used to set up a user heap.

---

void mem_rchp(char *area)

Free an area of common heap previously allocated via **mem_achp() .** Returns no errors. It always succeeds unless the parameter points to an invalid address, in which case the machine nearly always crashes!

---

char *mem_rlst (char **ptr, long *len)

Allocate a user area from an allocated area of common heap. 'ptr' is a pointer to a pointer to free space, len is the length requested to put in the user heap, and returns as the length actually allocated. Returns the address of the area allocated on success, and the SMS error code on failure.

---

chanid_t opw_con(WINDOWDEF_t *wdef)

Open a console window. The WINDOWDEF_t structure is defined in sys_qlib.h. Returns SMS channel id on success; SMS error code (which is negative) on failure.

---

chanid_t opw_scr (WINDOWDEF_t * windef)

Open a screen window. The WINDOWDEF_t structure is defined in sys_qlib.h. Returns channel on success, SMS errror code (which is negative) on failure.

---

chanid_t opw_wind (char *name, char *details)

Open a window. the 'name' parameter is a C type string that specifies the type and dimensions. The details parameter specifies the border width and colour and the paper/ink colours. Returns the SMS channel id on success and a SMS error code (which is negative) on failure.

---

void * sms_acnp(long size, long *sizegot, long jobid)

Allocate memory from common heap. Is passed requested size, plus job id which is to own the heap. Returns address of area allocated, or a QDOS error code on failure.

Note that even if the call succeeds this will not be the same as the size requested, as the amount requested is rounded up to the nearest 16 bytes and then the length of the common heap header is added on to it. If you are not interested in the true size obtained, then set 'sizegot' to NULL. Otherwise set it to the address of a variable that will be set to contain the actual size obtained.

---

void * sms_achp_acsi(long size, long *sizegot, long jobid)

Special variant of the sms_achp() call that is only relevant to Atari TT (or similar systems that support fast RAM). This call is guaranteed to use ST compatible RAM and NOT allocate the memory in fast RAM. This is important if the memory is to be used for ACSI/DMA purposes.

On all other systems this call is functionally identical to the standard sms_achp() call.

---

int sms_acjb(long jobid, unsigned char priority,
timeout_t timeout)

Start a activate a job with a given priority. There are two valid values for the timeout, 0 and -1. Execution of the current job will continue if the timeout is set to zero, and the SMS error code for this call returned. If the timeout is -1 then the current job is suspended until the activated Job has finished. This call will then return the error code from that Job.

---

void * sms_alhp (void **ptr, long *len)

Allocate a user area from an allocated area of common heap. 'ptr' is a pointer to a pointer to free space, len is the length requested to put in the user heap, and returns as the length actually allocated. Returns the address of the area allocated on sucess, and the SMS error code on failure.

---

void * sms_arpa(long size)

Allocate memory from resident procedure area. Returns address of area allocated, or a QDOS error code on failure. On QDOS systems, will always fail if called while any program except SuperBasic is executing.

---

void sms_artc(long sms_time)

Adjust the clock by sms_time seconds.

---

void sms_cach (long flag)

Change cache state. If 'flag' is 0, then cache is turned off and if it is 1 then the cache is turned on. Future releases of SMSQ and SMSQ/E may support alternative values for the 'flag' parameter for different cache control strategies.

NOTE. This call only works on SMSQ and SMSQ/E based systems.

---

void sms_comm (int rate)

Set the baud rate for both serial ports.

---

jobid_t sms_crjb(long codespace, long dataspace,
void *start_address, jobid_t owner,
void **job_address)

Create another job in the transient program area, given size of new jobs code, data the start address of the new job, and its owner. Returns either positive job id of new job, or SMS error code. Also returns address of newly created job in last parameter.

---

void sms_dmod(short * s_mode, short * d_type)

Set/read display mode.

*s_mode = 4 for mode 4,
= 8 for mode 8,
= -1 for read

*d_type = 0 for monitor mode,
= 1 for TV mode,
= -1 for read

Notes:

1) Other values are available for use in these parameters on Minerva

ssytems - refer to the Minerva documentation for details

2) There is a bug in QL roms that corrupts the return d_type when it is read.

---

int sms_exv (QLVECTABLE_t * table, long jobid)

Change the exception vector table for a particular job. The QLVECTABLE_t structure is defined in sys_qlib.h Returns SMS error code.

---

void sms_fprm(long * lang_code, char ** car_reg,
long * group_mod)

Find preferred module of the type and group requested. The 'lang_code' or 'car_reg' parameters can be zero.

NOTE. Only available on SMSQ and SMSQ/E based systems.

---

int sms_frjb(jobid_t jobid, int error_code)

Force remove a job, giving an error code for it to return. Returns SMS error code. If applied to the current job, then it will never return.

---

long sms_frtp ()

Find largest contiguous are available for loading a program. This is normally also a good indicator of the total free memory in the machine.

#include <things.h>

int sms_fthg (char * thing_name, jobid_t jobid, long * d2,
long d3, void * a1, void **a2)

Free the named 'thing'. Available as standard wiht SMS, and on QDOS compatible systems with THING support code loaded. Returns the SMS/QDOS error code. The parameters d2, d3, a1 and a2 are used to pass extra parameters as defined in the definition of the 'thing' that is being freed. Note also that the d2 and a2 parameters are pointers to these values as new values can be passed back from the 'thing' being freed. The d3 and a1 parameters are not changed, so pointers are not used for these parameters.

---

int sms_hdop(void * param_list)

Send a command to the 8049 second processor. Uses INTEL byte format (low byte first). Returns value returned by 8049.

---

jobid_t sms_info(void ** system_variables,
long * version_code)

Get the address of the system variables and the current operating system version code (in the form xx.xx - non zero terminated string). Returns job id of current job.

---

int sms_injb(jobid_t * jobid, jobid_t * topjob,
long * job_priority, void ** job_address)

Get information on a job within a job tree. Passed the jobid you want information on and the current top of the job tree you are looking at (with the first call set *topjob = *jobid). It is designed to be called repeatedly without changing jobid and topjob until *jobid == 0. Returns:

0 OK and 'job_address' contains address of job, 'jobp' contains job priority in least significant byte, and if the job is suspended the most significant byte is negative. 'jobid' and 'topjob' are changed to those of the next job in the tree.

-ve SMS error code.

---

void sms_iopr (short priority)

Set I/O priority.

NOTE. This call only works on SMSQ and SMSQ/E based systems.

---

void sms_lenq (long * lang_code, char car_registration[4])

Language enquiry. If both 'lang_code' and 'car_registration' parameters contain zeroes then the current settings for these values are returned.

Alternatively if you specify either the 'lang_code' or the 'car_registration' parameter as non-zero then the corresponding value for the other parameter will be returned. Note that the 'car registration' parameter is unusual in that it is a fixed length character field of length 4 that is always space filled.

The current language code is not changed.

NOTE. This call only works on SMSQ and SMSQ/E based systems.

---

void sms_lexi(QL_LINK_t * lnk)

Link in external interrupt handler

void sms_lfsd(QLDDEV_LINK_t * lnk)

Link in directory I/O device handler

void sms_liod(QLD_LINK_t * lnk)

Link in simple I/O device handler

void sms_lpol(QL_LINK_t * lnk)

Link in polled task handler

void smslshd(QL_LINK_t * lnk)

Link in scheduler list handler

The QL_LINK_t, QLD_LINK_t and QLDDEV_LINK_t structures are defined in sys_qlib.h

---

void sms_lldm (void * land_module)

Link in Language Dependent Module.

NOTE. Only available on SMSQ and SMSQ/E based systems.

---

void sms_lset (long * lang_code, char car_registration[4])

Language set. If both 'lang_code' and 'car_registration' parameters contain zeroes then the current settings for these values are returned.

Alternatively if you specify either the 'lang_code' or the 'car_registration' parameter as non-zero then the corresponding value for the other parameter will be returned. Note that the 'car registration' parameter is unusual in that it is character field with a fixed length of 4 that is always space filled.

The current language code is changed to the value that is returned in the 'lang_code' parameter. If no corresponding language code can be found, then the default language (the first language preference linbked in via sms_lldm) is set.

NOTE. This call only works on SMSQ and SMSQ/E based systems.

---

int sms_lthg (THING_LINKAGE * thing_linkage)

Link in a new Thing. Available as standard with SMS, and on QDOS compatible systems with THING support code loaded. The structure THING_LINKAGE is defined if you include the qdos.h or sms.h header files.

---

void * sms_mptr (long message_code)

Find message pointer. The message code value can be:

a) An address with the MSB set

b) The message group + message number (negated)

It returns a pointer to the message (or to the "unknown error" message).

NOTE. Only works on SMSQ and SMSQ/E based systems. On other systems a negative value is returned which is the QDOS/SMS error code.

#include <things.h>

int sms_nthg (char * thing_name, THING_LINKAGE **next_thing)

Find next Thing. Available as standard with SMS, and on QDOS compatible systems with THING support code loaded. The 'thing_name' parameter is a C style NULL terminated string. The 'next_thing' parameter is used to return the Thing Linkage block for the next Thing, or 0 if no further Thing exists. The THING_LINKAGE structure is defined in the sms.h header file. Returns SMS error code.

#include <things.h>

int sms_nthu (char *name, THING_LINKAGE ** thing_linkage, jobid_t * owner_job)

Get the owner of a job, and the next linkage block. If the pointer pointed to by thing_linkage is 0, then this the value returned in 'owner_job' is undefined, and this routine functions like the **sms_nthg()** routine.

---

void sms_rchp(void *area)

Free an area of common heap previously allocated. Returns no errors. It either succeesds or crashes if given an invalid area address.

---

void sms_rehp(void * area, void ** ptr, long len)

Link an area back into a user heap area. Given area to link in, pointer to pointer to free space, and length to link in. This call is also used to set up a user heap.

---

void sms_rexi(QL_LINK_t * lnk)

Unlink external interrupt handler

void sms_rfsd(QLDDEV_LINK_t * lnk)

Unlink directory I/O device handler

void sms_riod(QLD_LINK_t * lnk)

Unlink simple I/O device handler

void sms_rpol(QL_LINK_t * lnk)

Unlink polled task handler

void sms_rshd(QL_LINK_t * lnk)

Unlink scheduler list handler

The QL_LINK_t, QLD_LINK_t and QLDDEV_LINK_t structures are defined in sys_qlib.h

---

int sms_rmjb(jobid_t jobid, int error_code)

Remove a suspended job, giving an error code for it to return. Returns SMS error code.

---

long sms_rrtc (void)

Read clock. Returns time in seconds from Jan 1 1961.

---

int sms_spjb (long jobid, int new_priority)

Set the priority of a job. Sets current jobs priority if jobid = -1. Returns old priority of this job or a SMS error code.

---

int sms_rrpa (void * area)

Release an area of the resident procedure area previously allocated. Returns QDOS/SMS error code.

On QDOS systems, will always fail as only allowed when SuperBasic is the only program running. Other systems such as Minerva and SMSQ have released this restirction.

---

void * sms_schp (long size_wanted, long * new_size, void * old_base_address)

Shrink an allocation in the common heap. Returns SMS error code. The value returned will actually be the same as 'old_base_address' as the base address of the area will remain the same.

The new size will normally be larger than the size requested due to rounding effects within the operating system. If you want to know the exact value of the new size, then set the 'newsize' parameter to point to a variable that should be set to hold the new size. If you are not interested, then this parameter can be NULL.

NOTE. Only available on SMSQ and SMSQ/E based systems.

---

void sms_srtc(long sms_time)

Set the clock.

---

int sms_rthg (char * thing_name)

Remove a Thing if it is not in use. The 'thing_name' parameter is a C style (NULL terminated) string.

---

int sms_sevt (jobid_t jobid, event_t eventlist)

Send the events in 'eventlist' to the destination job. If the job is waiting on any of these events then the job is released, otherwise the events are discarded.

This call is only supported on SMSQ/E v2.71 or later. On other systems it will not be recognised, and an error reporting an unsupported trap value will be returned.

---

int sms_ssjb(jobid_t jobid, int number, char * zero)

Suspend a job for a number of 50Hz (or 60Hz if an American system) clock ticks. The 'zero' parameter is an address of a byte to set to zero on release of the job. If this is not required pass NULL as the value of the 'zero' parameter. If the 'number' parameter is -1 then the job is suspended indefinitely. Returns a SMS error code.

---

int sms_trns (const void * trans_table,
const void * msg_table)

Set the translate table and message table. This routine will not work on QL systems with ROMS that are of version JS or earlier. Returns SMS error code.

On SMSQ and SMSQ/E systems, if 'msg_table' parameter is not NULL, and the language code is $4afb, then this address is used for message group 0.

JOBHEADER_t * sms_usjb (jobId_t jobid)

Release a suspended job, sets _oserr, returns address of job header (the JOBHEADER_t structure is defined in sys_qlib.h) or QDOS error code.

#include <things.h>

char * sms_uthg (char * thing_name, jobid_t job_id,
timeout_t timeout, long *d2, void *a2,
long *version, THING_LINKAGE **linkage)

Use a Thing. The name is passed in C (NULL terminated) format. The version is returned in the 'version' parameter. The additional values passed/returned in the 'd2' and passed in the 'a2' parameters are dependent upon the definition of the THING being used. The 'linkage' parameter is used to get back the Thing linkage address on a successful call. If an error occurs, then the error code (which is negative) is returned. If successful, the address of the Thing is returned, and a pointer to its linkage in the 'linkage' parameter. The THING_LINKAGE structure is defined in the sms.h header file.

---

int sms_wevt (event_t *eventlist, timeout_t timeout)

Wait for one or more of the events in 'eventlist'. When the job is released, then 'eventlist' is updated to show which event(s) caused it to be released. If no events are indicated then the job was released because timeout occurred.

This call is only supported on SMSQ/E v2.71 or later. On other systems it will not be recognised, and an error reporting an unsupported trap value will be returned.

---

int sms_xtop()

NOT YET IMPLEMENTED.

This is reserved for a potential future implementation of the sms_xtop() routine used to extend the TRAP#1 series of calls.

NOTE. Only available on SMSQ and SMSQ/E based systems

#include <things.h>

int sms_zthg (char * thing_name)

Zap a thing. The name is supplied in C (NULL terminated) format. Returns SMS Error code.

---

void ut_werms(int qdoserror, chanid_t channel)

Write the error message corresponding to the given SMS error code to the specified channel.

---

void ut_wersy (int qdoserror)

Write the error message corresponding to the given SMS error code to channel 0.

---

int ut_wint(chanid_t channel, int value)

Convert a value to ASCII and send it to the specified channel. Returns SMS error code (if any).

---

int ut_wtext(chanid_t, QLSTR * message)

Send a message to a specified channel. Returns SMS error code (if any).

MANIFEST CONSTANTS

There follwoing maifest constants are defined in QODS.H for the error codes returned by QDOS.

Constant	Meaning
ERR_OK	NO error occured
ERR_BFFL	**B..F** for **F..L**

ERR_BFIL	**B** u**F** fer **F** u**L** l
ERR_DRFL	**DR** ive **F** u**L** l
ERR_EOF	**E** nd **O** f **F** ile
ERR_FDIU	**F** ile or **D** evice **I** n **U** se
ERR_FDNF	**F** ile or **D** evice **N** ot **F** ound
ERR_FEX	**F** ile already **EX** ists
ERR_FMTF	**F** or**M** a**T** **F** ailed
ERR_ICHN	**I** nvalid **CH** a**N** nel id
ERR_IEXP	**I** nvalid **EXP** ression
ERR_INAM	**I** nvalid file, device or thing **NAM** e
ERR_IJOB	**I** nvalid **JOB** id
ERR_IMEM	**I** nsufficient **MEM** ory
ERR_IPAR	**I** nvalid **PAR** ameter
ERR_ISYN	**I** nvalid **SYN** tax
ERR_MCHK	file system **M** edium **C**ª ecK failed
ERR_NC	operation **N** ot **C** omplete
ERR_NIMP	**N** ot **IMP** lemented
ERR_ORNG	**O** utside permitted **R** a**NG** e
ERR_OVFL	arithmetic **OV** er**FL** ow
ERR_PRTY	**P** a**R** i**TY** error
ERR_RDO	**R** ea**D** **O** nly
ERR_RWF	**R** ead or **W** rite **F** ailed/
ERR_TRNE	**TR** a**N** smission **E** rror

CHANGE HISTORY

30 Jul 93	Added SMS entry points into library. These just refer you to the QDOS name for the more detailed description unless the call is only available SMS and not under QDOS.
08 Sep 93	Added c_extop() call (based on a contribution by PROGS of Belgium).
31 Dec 93	Documented the iop_outl() call.
24 Jan 94	The LIBSMS_DOC file created to hold the details of the SMS Operating System Interfaces of C68. All call definitions expanded to avoid the need to cross-reference the same calls under their QDOS names. Details of names of manifest constants used for SMS error codes added.
08 Apr 95	Added the new calls that are only available on SMSQ and SMSQ/E based systems. Changed some of the existing definitions to use the more generic 'void *' for address parameters instead of the previous 'char *'.
04 Jan 96	Added descriptions of the various system vector calls available that were not previously documented. These have always been available in the library, but were previously only documented under their QDOS names.
03 May 96	Added new event handling trap calls introduced with SMSQ/E v2.71.
19 Jul 98	Added the ioa_cnam() routine definition.

Standard C library: UNIX routines

This section of the C68 library documentation covers those routines in the C68 Standard C library that are marked as providing UNIX compatibility (except for any routines that use the math.h header file as these are documented in LIBM_DOC file).

Please note, however, that it cannot be guaranteed that all versions of Unix support the routines listed in this category. Except where specified otherwise Unix SVR4 is taken as the base line. Also fo historical reasons many non-Unix systems support many of the routines listed under this category.

---

void _exit (int status)

Exit the program with the specified status. Does NOT first do a tidy close on open files.

Defined in unistd.h

---

int access(char *name)

Defined in fcntl.h

---

int alarm (unsigned int seconds)

Unix and Posix compatible routine to send the calling process a SIGALARM signal after the specified number of seconds. Returns 0 if there was not already an alarm outstanding, or the amount of time left on the previous alarm if there was one. If seconds is specified as 0, then any pending alarm is cancelled.

Note that normally this routine is defined as returning an "unsigned int" value as no error can occur. In the QDOS/SMS implementation of signals it IS possible for an error return to happen (which will be negative).

Defined in unistd.h

---

int allmem(void)

Defined in stdlib.h

---

char *argopt(int argc, char *argv[], char *opts,
int *argn, char *optc)

Gets next argument from list.

Defined in stdlib.h

---

char * basename (char * pathname)

Treats the string supplied as a pathname, and returns a pointer to the filename part following any directory part.

Note that as QDOS level 1 systems do not support true directories a simple heuristic is used that may occasionally give excentric results. In particular it assumes that the extension part of a filename cannot be longer than 4 characters. It will however, provide results consistent with the dirname() function.

Needs linking with **-lgen** to include this function

Define in libgen.h

---

int bcmp (char * string1, char * string2, len)

Compares two strings. Replaced under ANSI by memcmp().

Defined in memory_h

---

char *bcpy (char *source, char *target, int length)

Copies an area of memory. Replaced under ANSI by memcpy(). Note that operands in reverse order to memcpy().

Defined in memory_h

---

char * bgets (char * buffer, size_t *count, FILE * fp
const char * breakstring)

Unix compatible routine to read a file up to the next delimiter. The file is read until either count is exhausted or one of the break characters specified in 'breakstring' is encountered. The data is then terminated with a NULL byte, and a pointer to this byte returned.

Needs linking with **-lgen** to include this function

Defined in libgen.h

---

int bldmem(int n)

Defined in stdlib.h

---

size_t bufsplit (char * buf, size_t n, char **a)

Unix compatible routine to take a buffer containing fields separated by field seperators and setting up the 'a' array of pointers to point to each field.

The default set of field termiators are assumed to be tab and newline, but alternative separators can be specified by giving a list in the 'buf' parameter with the 'n' and 'a' parameters both set to 0.

The return value is then number of fields found and thus unique addresses set in the 'a' array. All other values of the 'a' array are set to point to the NULL byte at the end of the buffer.

Needs linking with **-lgen** to include this function

Defined in libgen.h

---

void bzero (char *target, int length)

Sets an area of memory to zero. Under ANSI the memset() function is used instead.

Defined in memory_h

---

int chdir(char *str)

Changes current data directory (as set by TK2 DATA_USE command in SuperBasic). If passed NULL or strings of the form "../" or "..", then tries to go up a level. If passed a string starting with a device name then replaces the current directory, else appends to current directory (adding _ at end if needed). Maximum length is 31 characters. Returns 0 if ok, !0 if failed.

(See also libqdos documentation for chddir() and chpdir())

Defined in stdlib.h

---

int chmod (char * filename, int access_modes)

Unix compatible routine for setting the file access permission bits. In the QDOS implementation, the Read and Write options are ignored as QDOS does not have the concept of "Read Only" or "Write only" access. The Execute bit is used to,determine whether the file should be marked as EXEC'able for normal files, and ignored for directories.

Defined in sys/stat.h

---

int chown (const char * path, uid_t owner, gid_t group)

Unix/Posix compatible call to change the owner and group of a file. Since QODS and SMS do not support the concepts of owners and groups for files, this call will always return as though it had completed successfully.

Defined in unistd.h

---

int close(int fd)

Closes a file. -1 if error, 0 if ok.

Defined in fcntl.h

---

int closedir (DIR *)

Close a directory.

Defined in dirent.h

---

void clrerr(FILE *fp)

Lattice version of the ANSI clearerr() routine

Macro defined in stdio.h

---

char * copylist (const char * filename, size_t * sizeptr)

Unix compatible routine to read a file into memory. The memory required is allocated using malloc(). All newlines in the file are changed to NULL bytes. If any error occurs, then NULL is returned. If successful, the size of the file is stored at the location pointed to by 'sizeptr' and the address of the memory allocated is returned.

Needs linking with **-lgen** to include this function

Define in libgen.h

---

int creat(char *name)

Routine to create a file. If the file exists it is truncated, if it does not it is created. Returns new fd or -1 if error.

Defined in fcntl.h

---

char * dirname (char * pathname)

Treats the string supplied as a pathname, and returns a pointer to the path part with any filename part removed.

Note that as QDOS level 1 systems do not support true directories a simple heuristic is used that may occasionally give excentric results. In particular it assumes that the extension part of a filename cannot be longer than 4 characters. It will however, provide results consistent with the baseame() function.

Needs linking with **-lgen** to include this function

Define in libgen.h

---

int dup(int fd)

Routine to duplicate at level 1 the number of file descriptors accessing the same file. Returns -1 on error, new file descriptor on success.

Defined in fcntl.h

---

int dup2(int fd, int nfd)

int fd - File descriptor to duplicate

int nfd - File descriptor to re-allocate

Routine to duplicate at level 1 the number of file descriptors accessing the same file, given the second descriptor to use explicitly (Closes it if already open). Returns -1 on error, 0 on success.

Defined in fcntl.h

---

void dqsort(double *da, int n)

Lattice compatible routine for sorting an arry of doubles into order.

Defined in stdlib.h

---

void endpwent (void)

Unix compatible routine to close the password file. As QDOS and SMS do not support the password file this merely simulates this action.

Defined in pwd.h

---

int envunpk (char * env)

Lattice compatible routine to take an array of environment strings and create an array of pointers to the strings. The address of the array is stored in the global variable 'environ' and the count of strings returned. The memory for the array is allocated dynamically, and that for any previous array automatically released. If an error occurs allocating memory, then -1 is returned.

Defined ins sdlib.h

---

int exec(....)

Routines to start off another process with a priority found in external variable **_def_priority** . Waits for this newprocess to finish and returns its QDOS error code. _oserr == 0 if error code is from new process, else process didn't start if _oserr != 0. Sets errno and _oserr.

Note that this differs from the traditional Unix style **exec()** family of calls in that it passes an additional parameter (as the second parameter) to define the channels to be passed. The ' **file_desc** ' parameter points to an array of file descriptors to pass to the new process. file_desc[0] is the number of file descriptors to follow, followed by the level 1 file descriptors in file_desc[1], file_desc[2],..... file_desc[chan[0] - 1]. If the 'file_desc' parameter is -1L, then the current programs file descriptors 0, 1, and 2 (stdin, stdout, stderr) will be passed.

The strings passed in either the argv array or the list of args must begin with a string containing the name of the program (this is UNIX convention).

eg. to exec a program test with arguments "this is a test" and keeping the same channels, Use :-

```
    execl( "test", -1L, "test", "this is a test", NULL);
```

The string "this is a test" will be parsed correctly into separate strings for argv[1] etc. by the receiving program.

The variants of **exec** available with C68 are:

int execv(char * name, int * file_desc, char * argv[])

int execvp(char * name, int * file_desc, char * argv[])

int execl(char * name, int * file_desc, char * argvs, ...)

int execlp(char * name, int * file_desc, char * argvs, ...)

The directory search sequence in each case is:

execv program directory only

execvp program directory, then data directory

execl program directory only

execlp program directory amd then data directory

The **execl** and **execlp** variants must have their parameter lists termianted by a NULL parameter.

Defined in unistd.h

---

void _exit (int exit_code)

Routine to close a program immediately without attempting to flush any open files. If an exit routine has been logged via onexit(), then this is called before quitting. The value of the 'exit_code' parameter is returned to the initiator of this program.

Defined in unistd.h

---

int fcntl(int fd, int action, int flags)

Routine to get and set various file parameters such as file flags, type of I/O device etc. returns -1 on error, various other things depending on the options used. These are defined in fcntl.h and are

F_GETFD - gets device type

F_SETFD - set device type

F_GETFL - returns device flags (as O_APPEND,O_RAW etc,

instead of internal values)

F_SETFL - sets device flags given O_APPEND etc.

Defined in fcntl.h

---

char *fcvt(double v, int dec, int *decx, int *sign)

Defined in fcntl.h

FILE *fdopen(int fd, char *mode)

Defined in stdio.h

---

int fdmode(int fd, int mode)

Routine to change raw or cooked mode using level 1 file. Returns -1 on error (with more info in errno) and 0 on success. If mode != 0 sets O_RAW on fd. if mode == 0 removes O_RAW on fd (no effect if not already set).

Defined in fcntl.h

---

int fgetchar(void)

Defined in stdio.h

---

long fgetchid(FILE *fp)

Returns QDOS channel id of FILE pointer. Returns -1L on error

Defined in stdio.h

---

int fileno(FILE *fp)

Macro.

Defined in stdio.h

---

int fflushall(void)

Defined in stdio.h

FILE *fopene(char *name, char *mode, int paths)

paths == 3 - search program directory, then data directory

paths == 2 - just search program directory

paths == 1 - search data directory first, then program

directory

paths == 0 - just search data directory (as open())

Defined in stdio.h

---

int fork(...)

Starts another process concurrently that is owned by this process. The new process is started with a default priority found in external variable **_def_priority** . Returns process id of new process or error code. Sets errno (and if relevant _oserr). The arguments have the same meaning as in the **exec()** family of calls.

Note that the semantics of the **fork()** family of calls in QDOS are different to that of Unix systems. This is an unavoidable consequence of the fact that QDOS/SMS systems have no memory management hardware. Therefore, such calls in any source code being ported will always need examining carefully to work out how to achieve the desired effect.

Note also that there is also the **qfork()** family of calls (defined in LIBC68_DOC) that is functionally similar to these calls except that a specified job can be made into the owner. The **qfork()** versions hsould be used when you do not want the daughter job to die if the parent job terminates.

pid_t forkv(char * name, int * file_desc, char * argv[])

pid_t forkvp(char * name, int * file_desc, char * argv[])

pid_t forkl(char * name, int * file_desc, char * argvs, ...)

pid_t forklp(char * name,int * file_desc, char * argvs, ...)

The directories searched in each case are as follow:

The directories searched in each case are as follow:

forkv program directory only

forkvp program directory and then data directory

forkl program directory only

forklp program directory and then data directory

The **forkl()** and **forklp()** routines must have a NULL parameter to terminate their parameter lists.

Defined in stdlib.h

---

long fpathconf (int filedes, int name)

Posix compatible routine to get configuration information on a file or directory. This is currently a dummy routine, and will always return an error code.

Defined in unistd.h

---

int fputchar(int c)

Defined in stdio.h

---

void fqsort(float *fa, int n)

Lattice compatible routine for sorting an array of floats into order.

Defined in stdlib.h

---

int fstat (int fd, struct stat *buf)

Unix compatible routine to get file status information for a level 1 file. One difference between Unix and the QDOS C68 implementation is that only directory devices are supported. Any attempt to use **fstat()** on a non-directory device will result in a ENOTBLK error being generated.

Returns:

0 successful

-1 error occured - errno set to indicate type.

Defined in sys/stat.h

---

int fsync(int fd)

Routine to flush a level 1 file. Returns -1 on error, 0 on success.

Defined in fcntl.h

---

int ftruncate (int file_descriptor, off_t length)

Unix compatible routine to truncate an open file to a specified length.

Defined in unistd.h

---

int getch(void)

Lattice compatible routine to get a character from the console without echo. If a console channel is currently open it will be used, but if not a new one will be opened. Note that no attempt is made to activate a cursor.

Defined in stdio.h

---

int getche(void)

Lattice compatible routine to get a character from the console with echo. If a console channel is currently open it will be used, but if not a new one will be opened. Note that no attempt is made to activate a cursor.

Defined in stdio.h

---

char *getcwd(char *str, int size)

Gets current data directory path (as set by TK2 DATA_USE command) into buffer str. If str == NULL then allocates a buffer of length size using malloc and returns address of it. Returns NULL on error, else address where name is stored. See also getcdd() and getcpd() in QDOS specific section.

Defined in unistd.h

---

gid_t getegid (void)

Unix compatible routine to get effective group id. Actually a dummy that always returns a value equivalent to the Unix user **root** .

Defined in stdlib.h

---

uid_t geteuid (void)

Unix compatible routine to get effective user id. Actually a dummy that

always returns a value equivalent to the Unix user **root** .

Defined in stdlib.h

---

gid_t getgid (void)

Unix compatible routine to get group id. Actually a dummy that always returns a value equivalent to the Unix user **root** .

Defined in stdlib.h

---

char *getmem(int size)

Defined in stdlib.h

---

char *getml(long size)

Defined in stdlib.h

---

int getopt(int argc, char * argv[], char *option_string)

Unix compatible option to help with parsing a command line that conforms to Unix syntax. It gets the next option from argv array that matches a letter in option_string. The option_string contains the letters that are valid parameter options to a C program. If the letter is followed by a colon, then this indicates that the option takes an argument (typically a filename). A number of external variables are used/set by the getopt() routine as follows:

char *optarg /* pointer to option argument */

int optind /* argv index of next argument */

int opterr /* set 0 stops message if parse fails */

int optopt /* character that caused the error */

The value returned by getopt() is the value of the option character, or '?' if an error occurred.

Defined in stdlib.h

---

char * getpass(char * prompt)

Unix/Posix compatible routine to read a password from a controlling terminal.

This routine will try and use the channel associated with stderr as long as it is a console channel (Unix/Posix would use /dev/tty). Characters will be read until either EOF or a newline occurs. A pointer to the NULL terminated string read will be returned, or NULL if an error occurred. If the string entered is longer than PASS_MAX (defined in limits.h) then it will be truncated to that length. Note that an internal buffer is used that is overwritten each time this routine is called.

Defined in stdlib_h

---

long getpid()

Routine to get the QDOS job id of the current job. Sets _oserr and returs either job id or QDOS error code.

Defined in stdlib.h

---

struct passwd *getpwent (void)

Returns a pointer to a an object of type 'struct passwd'. Because QDOS/SMS do not support the concept of a password file, this call simluates the existence of password file containing a single entry belonging to the "root" user (this is the Unix super-uer).

Defined in pwd.h

---

uid_t getuid (void)

Unix compatible routine to get user id. Actually a dummy that always returns a value equivalent to the Unix user **root** .

Defined in stdlib.h

---

int iabs(int i)

Lattice compatible routine to compute the absolute value of an integer.

Macro defined in stdlib.h

---

char *index (char *string, int c)

Search string for occurrence of a character. This is now an obsolete routine that under ANSI has been superseded by the strchr() function.

Defined in string.h

---

int iomode(fd, mode)

int fd, mode;

Routine to change the i/o mode of a level 1 file. Exclusive OR's the current flags in the ufb structure with the passed flags. Returns the previous value of the flags, or -1 if error, so if 0 is passed in the mode field then no change is made to the flags field, it is just returned unchanged.

ufb structure and flags defined in fcntl.h

Defined in fcntl.h

---

int isascii(int c)

Macro in ctype.h (or function if ctype.h not included)

Defined in ctype.h

---

int isatty(int fd)

Unix compatible routine to find out if a file descriptor corresponds to a tty (**con** or **scr** under QDOS) device. Returns 0 if not, 1 if it is

Defined in stdlib.h

---

int iscsym(int c)

int iscsymf(int c)

Macros in ctype.h (or function if ctype.h not included)

Defined in ctype.h

---

char * itoa (int number, char * target)

Converts a number into an ASCII string and returns address of the string. It is the user's responsibility to ensure that the target string is large enough to hold the result.

Defined in stdlib.h

NOTE. Many systems do not support this routine (It is not part of ANSI, POSIX or Unix SVR4 definitions). For maximum compatibility you should use the sprintf() function instead (or write your own itoa() function).

---

int kbhit( )

Lattice compatible routine to detect any pending keypresses on the console. Returns 0 if none, or no current console channel, 1 if there is input waiting.

Defined in stdio.h

---

int kill (pid_t program_id, int signal_numer)

int killu (pid_t program_id, int signal_numer, int uval)

The kill() routine is a Unix and Posix compatible routine to send a signal to a job.

The killu() variant is oen that is sometimes encountered that allows an additional parameter to be passed.

On success returns 0. On failure returns -1 and sets errno.

Defined in signal.h

---

struct lconv *localeconv (void)

Posix comaptible routine to get the current settings of the locale dependent information as contained in the 'lconv' structure.

Defined in locale.h

---

int link (const char *path1, const char * path2)

Unix compatible routine to link a file (i.e. give it an alternative name). QDOS and SMS do not support the concept of links, so this call will always fail with an error indicating that the maximum number of links have been exceeded for the file in question.

Defined in unsitd.h

---

void lqsort(long *la, int n)

Lattice compatible routine for sorting an array of long integers into ascending order.

Defined in stdlib.h

---

char *lsbrk(long size)

Defined in stdlib.h

---

long lseek(int fd, long offset, int mode)

long tell(int fd)

ftell is a macro in fcntl, equal to lseek(fd, 0L, 1)

Seeks to correct position within file fd. offset specifies position, mode is either SEEK_START - absolute, SEEK_REL - relative, SEEK_END - relative to eof (position must be -ve or zero). Returns new position or -1 on error.

Defined in fcntl.h

---

char *memccpy(char *to, char *from, int c, size_t n)

Defined in string.h

---

int mkdir(char * name)

Unix compatible routine to make a directory. Returns 0 on success, -1 on error (and sets errno). This will only succeed on systems that support Version 2 (or later) filing systems.

Defined in sys/stat.h

---

int mkfifo (const char * name, mode_t mode)

Posix compatible routine to create a FIFO. This is currently a dummy and will always return an error.

Defined in sys/stat.h

---

int mknod (const char * path, mode_t mode, dev_t dev)

Unix compatible routine to make a directory, a standard file or a special file. Because QDOS and SMS do not support special file types (in the Unix sense) any attempt to create a file of this type will result in an error return.

Defined in sys/stat.h

---

char *mktemp (char * template)

Unix compatible routine to generate a unique filename. The template is a string of the form "filenameXXXXXX". The 'X's will be overwritten with characters that generate a unique filename. Returns a pointer to this string.

Defined in stdlib.h

---

void movmem(char *from, char *to, unsigned int n)

Move memory. ANSI form is memmove().

Macro defined in string.h

---

int onexit(int (*function)())

Defined in stdlib.h

---

int open(char *name, int mode)

General UNIX compatible routine to open a file.

Defined in fcntl.h

DIR *opendir(char * directoryname)

Unix compatible routine to open a directory.

Defined in dirent.h

---

int opene(char *name, int mode, int paths)

Routine to search more than just the default

directory, if name does not start with a device. If it does then that is opened, else if :-

paths == 3 - search program directory, then data directory

paths == 2 - just search program directory

paths == 1 - search data directory first, then program directory

paths == 0 - just search data directory (as open())

return -1 on error, valid fd if ok.

Defined in fcntl.h

---

long pathconf (char * path, int name)

Posix compatible routine to get configuration information on a file or directory. This is currently a dummy routine, and will always return an error code.

Defined in unistd.h

---

int pause(void)

Posix compatible routine to suspend a process until a signal received. There is no successful return, so it always returns -1 and sets errno to indicate why it returned.

Defined in unistd.h

---

int pclose (FILE *fp)

Close a pipe opened by **popen()** after waiting for the associated process to finish. Returns exit status of process, or -1 if their is no associated process.

Defined in stdio.h

---

int pipe(fdp)

int fdp[2]; /* Two file descriptors, fdp[0] = I/P pipe, fdp[1] = O/P pipe */

Routine to allow input and output pipes to be constructed, connected to each other. Both owned by the present job Returns -1 on error, 0 on success. Default size of o/p pipe is defined in external variable long _pipesize.

Defined in unistd.h

FILE * popen (char *command, char *type)

Create a pipe between the calling program and the command to be executed. Command is a C string giving the command (and any associated parameters) to be executed. 'type' is either "r" for reading or "w" for writing. The value returned by popen is a file pointer such that one can write to the standard input of the command if the type is "w", and read from its standard output if type was "r". Returns NULL if the file or associated process cannot be created.

Defined in stdio.h

---

int putch (int c)

Lattice compatible routine to output a character to the console. If there is no console channel open, then using this routine will cause one to be opened.

Defined in stdio.h

---

int raise (int signal_number)

int raiseu (int signal_numer, int uval)

The raise() routine is a Unix compatible routine to allow a process to send a signal to itself.

The raiseu() function is a variant that is occasionally encountered that allows an additional parameter to be passed.

The fraise() function is a C68 version that tries to raise the signal disregarding any blocking condition. It is intended for use from mt_trapv type routines. Does not work when called in SV mode.

On success returns 0. On failure returns -1 and sets errno to indicate reason.

Defined in signal.h

---

void rbrk(void)

Defined in stdlib.h

---

long read(int fd, char *buf, long size)

Reads size bytes into array at buf from file fd. size may be greater than 32K. Returns number of bytes read or -1 if error.

Defined in fcntl.h

---

struct dirent *readdir(DIR *direptr)

Unix compatible routine to read a directory.

Defined in dirent.h

---

int unlink(char *name);

Both delete a file. 0 if ok, -1 if error.

Defined in fcntl.h

---

void repmem(char *to, char *vt, int nv, int nt)

Defined in string.h

---

void rewinddir(DIR *dirptr)

Unix compatible routine to reset to directory start.

Macro defined in dirent.h

---

char * rindex (char * string, int c)

Search string for last occurence of a character. This is now an obsolete variant that under ANSI has been superseded by the strrchr() function.

Defined in string.h

---

int rlsmem(char *p, short n)

Lattice C compatible routine for releasing a memory allocation.

Defind in stdlib.h

---

int rlsml(char *p, long n)

Lattice C compatible routine for releasing a memory allocation.

Defined in stdlib.h

---

int rmdir (const char * directory_name)

Unix compatible call to remove a directory. The directory must be empty or the call will fail. This call will only work on QDOS or SMS systems that have support for hard directories.

Defined in unistd.h

---

void rstmem(void)

Lattice C compatible routine for resetting the memory allocation system.

Defined in stdlib.h

---

char *sbrk(unsigned int n)

Unix compatible memory allocation routine

Defined in stdlib.h

---

void seekdir(DIR *dirptr, long location)

Unix compatible routine to return the current position in a directory.

Defined in dirent.h

---

int seteuid (uid_t uid)

Unix compatible routine to set effective user id. Actually a dummy under QDOS that has no affect.

Defined in stdlib.h

---

int setuid (uid_t uid)

Unix compatible routine to set user id. Actually a dummy under QDOS that has no affect.

Defined in stdlib.h

---

void setmem(char *to, unsigned n, char c)

Defined in string.h

---

int setpgrp (void)

Unix compatible routine to set program group. Actually a dummy under QDOS that has no affect.

Defined in stdlib.h

---

void setpwent(void)

Unix compatible routine to reset to start of password file. As QDOS and SMS do not support the concept of a password file this merely simulates this action.

action.

Defined in pwd.h

---

int setnbf(FILE *fp)

Set a stream to be unbuffered

Defined in stdio.h

---

int setuid (uid_t uid)

Unix compatible routine to set user id. Actually a dummy under QDOS that has no affect.

Defined in stdlib.h

---

int sigaction (int signal_number,
struct sigaction * new_action,
struct sigaction * old_action)

Posix compatible routine for examining and changing signal actions. If 'new_action' is not NULL, it points to a structure defining the action to be associated with the specified signal. If the 'old_action' field is not NULL, then it must point to a structure where the previous action can be stored. The structure 'sigaction' is defined in teh signal.h file. On success returns 0. On failure returns -1 and sets errno.

Note that this routine is intended by Posix to supersede the signal() routine commonly used on Unix systems.

Defined in signal.h

---

int sigaddset (sigset_t * signal_set, int signal_number)

Posix compatible routine to add a signal to a signal set. Returns 0 on success, -1 on failure (and sets errno).

Defined in signal.h

---

int sigdelset (sigset_t * signal_set, int signal_number)

Posix compatible routine to Delete a signal from a signal set. Returns 0 on success, -1 on failure (and sets errno).

Defined in signal.h

---

int sigemptyset (sigset_t * signal_set)

Posix compatible routine to initialise and empty a signal set. Returns 0 on success, -1 on failure (and sets errno).

Defined in signal.h

---

int sigfillset (sigset_t * signal_set)

Posix compatible routine to initialise and fill a signal set. Returns 0 on success, -1 on failure (and sets errno).

Defined in signal.h

---

int sighold (int signal_number)

Unix compatible routine to set a signal to be blocked. On success returns 0, on failure returns -1.

Defined in signal.h

---

int sigignore (int signal_number)

Unix compatible routine to set a signal to be ignored. On success returns 0, on failure returns -1

Defined in signal.h

---

int sigismember (sigset_t * signal_set, int signal_number)

Posix compatible routine to test for a signal in a signal set. Returns 0 on success, -1 on failure (and sets errno).

Defined in signal.h

---

int siglongjmp (sigjmp_buf, int val)

Posix compatible routine. Version of longjmp() that preserves signal status as well.

Defined in setjmp.h

---

void (*signal(int signal_number, void (*sig_func)(int)))(int)

Unix compatible routine to set a signal handler (one of the nastier constructs in C!). Now superseded by the Posix defined routine sigaction(), but still commonly found in Unix originated programs.

Defined in signal.h

---

int sigpause (int signal_number)

Unix compatible routine to remove a signal and suspend. On success returns 0, on failure returns -1.

Defined in signal.h

---

int sigpending (sigset_t * signal_set)

Posix compatible routine to examine pending signals. Store the current list of pending signals into 'signal_set' Returns 0 on success, -1 on failure (and sets errno).

Defined in signal.h

---

int sigprocmask (int action, sigset_t * new_signal_set,
sigset_t * old_signal_set)

Posix compatible routine to examine/change blocked signals. The action is one of SIG_SETMASK,SIG_BLOCK,SIG_UNBLOCK. If 'old_signal_set' is not a NULL pointer, the previous mask is stored in the location to which it points. If 'new_signal_set' is not a NULL pointer then the signal mask to which it points is used to change the currently blocked set. If it is a NULL pointer, then the value of the 'action' parameter is irrelevant, and the mask is unchanged.

Returns 0 on success, -1 on failure (and sets errno).

Defined in signal.h

---

int sigrelse (int signal_number)

Unix compatible routine to remove a signal from the processes signal mask. On success returns 0, on failure returns -1.

Defined in signal.h

---

void (*sigset(int signum, void (*disp)(int)))(int)

Unix compatible routine for setting signals. Similar to signal() in most cases. You will need to examine Unix documentation for the differences.

Defined in signal.h

---

int sigsetjmp (sigjmp_buf, int savemask)

Posix compatible routine. Version of setjmp() that preserves signal status as well as long as savemask is non-zero. If savemask is zero, functions just like a normal setjmp().

Defined in setjmp.h

---

int sigsuspend (sigset_t * signal_mask)

Posix compatible routine to wait for a signal. Returns 0 on success, -1 on failure (and sets errno).

Defined in signal.h

---

long sizmem(void)

Lattice C compatible for determining available size of memory allocation pool.

Defined in stdlib.h

---

unsigned int sleep (unsigned int period)

Unix compatible routine to wait for a specified number of seconds.

Defined in unistd.h

---

void sqsort(short *sa, int n)

Defined in stdlib.h

---

int stat (char *filename, struct stat *buffer)

Unix compatible routine for getting file status.

Returns:

0 Success

-1 Error occured - errno set to indicate the type.

Defined in sys/stat.h

---

int stcarg(char *s, char *b)

Defined in string.h

---

char *stccpy(char *to, char *from)

Defined in string.h

---

int stcd_i(char *in, int *ival)

Defined in string.h

---

int stcd_l(char *in, long *lval)

Defined in string.h

---

int stch_i(char *in, int *ival)

Defined in string.h

---

int stch_l(char *in, long *lval)

Defined in string.h

---

int stcis(char *s, char *b)

Defined in string.h

---

int stcisn(char *s, char *b)

Defined in string.h

---

int stci_d(char *out, int ival)

Defined in string.h

---

int stci_h(char *out, int ival)

Defined in string.h

---

int stcl_d(char *out, long lval)

Defined in string.h

---

int stcl_h(char *out, long lval)

Defined in string.h

---

int stcl_o(char *out, int ival)

Defined in string.h

---

int stclen(char *s)

Lattice compatible routine to get length of a string. Functionally equivalent to strlen() routine.

Defined in string.h

---

int stco_i(char *in, int *ival)

Defined in string.h

---

int stco_l(char *in, long *lval)

Defined in string.h

---

int stcpm(char *s, char *patt, char **match)

Defined in string.h

---

int stcmpa(char *string, char *pat)

Defined in string.h

---

int stcu_d(char *out, unsigned int uval)

Defined in string.h

---

int stcul_d(char *out, unsigned long lval)

Defined in string.h

Defined in string.h

---

int stime (const time_t *tp)

Unix compatible routine for setting the system time.

Defined in unistd.h

---

char *stpblk(char *s)

Lattice compatible routine to skip blanks (white space). Returns pointer to first non-whitespace character, or to NULL byte at end of string.

Defined in string.h

---

char *stpbrk(char *s, char *b)

Obsolete Lattice variant of ANSI strpbrk() function.

Macro defined in string.h

---

char *stpchr(char *s, char c)

Obsolete Lattice variant of ANSI strchr() function.

Macro defined in string.h

---

char *stpchrn(char *s, char c)

Obsolete Lattice variant of ANSI strrchr() function.

Macro defined in string.h

---

char *stpcpy(char *to, char *from)

Defined in string.h

---

char *stpdate(char *p, int mode, char *date)

Defined in string.h

---

char *stpsym(char *s, char *sym, int symlen)

Defined in string.h

---

char *stptime(char *p, int mode, char *time)

Defined in string.h

---

char *stptok(char *s, char *tok, int token, char *brk)

Defined in string.h

---

int strbpl(char *s[], int max_array_size, char *string_list)

Lattice compatible routine to build an array of pointers to strings given a list of NULL byte terminated strings, with the list itself also terminated by a NULL byte. Returns the cound of strings, or -1 if not enough space in pointer array.

Defined in string.h

---

char *strcadd (char *target, const char * source)

Unix compatible routine for copying a string compressing any embedded C language escape sequences to the equivalent character. Returns a pointer to the NULL byte that terminates the string. If this routine is used, then a **-lgen** parameter must be used at link time.

Defined in libgen.h

---

char *strccpy (char *target, const char * source)

Unix compatible routine for copying a string compressing any embedded C language escape sequences to the equivalent character. Returns a pointer to the start of the target string. If this routine is used, then a **-lgen** parameter must be used at link time.

Defined in libgen.h

---

char *strdup(char *s)

Defined in string.h

---

char *streadd (char *target, const char * source,
const char * exceptions)

Unix compatible routine for copying a string expanding any non-grpahics

characters into embedded C language escape sequences. Returns a pointer to the NULL byte that terminates the string. The target area must be large enough to hold the resultant string. In a worst case scenario, 4 times the size of the source area is guaranteed to be large enough. The exceptions parameter is used to specify any chanracters that must be passed through unchanged. A NULL can be used if there are no exceptions. If this routine is used, then a **-lgen** parameter must be used at link time.

Defined in libgen.h

---

char *strecpy (char *target, const char * source,
const char *exceptions)

Unix compatible routine for copying a string expanding any non-graphic characters into embedded C language escape sequences. Returns a pointer to the start of the target string. The target area must be large enough to hold the resultant string. In a worst case scenario, 4 times the size of the source area is guaranteed to be large enough. The exceptions parameter is used to specify any chanracters that must be passed through unchanged. A NULL can be used if there are no exceptions. If this routine is used, then a **-lgen** parameter must be used at link time.

Defined in libgen.h

---

int strfind(char *tosearch, char *tofind)

Find the position of string 'tofind' in string 'tosearch'. Returns -1 if not found, position in string if found. If this routine is used, then a **-lgen** parameter must be used at link time.

Note. If you want a version that does case independent matching, then use strfnd() instead (defined in LIBC68_DOC).

Defined in libgen.h

---

int stricmp (const char * string1, const char * string2)

Do a case independent compare of two strings. Return 0 if they match.

Defined in string.h

---

void strins(char *to, char *from)

Lattice compatible routine to insert one string in front of another to produce a larger string.

Defined in string.h

---

char *strlwr(char *s)

Lattice compatible routine to convert a string to lower case.

Defined in string.h

---

int strnicmp(char *a, char *b, int n)

Do a length limited case independent compare of two strings. Return 0 if they match.

Defined in string.h

---

char *strnset(char *s, int c, int n)

Do a length limited initialisation of a string to a specified character value. Return the address of teh string.

Defined in string.h

---

int strpos (char * string, int c)

Returns position of first occurrence of character c in the string, and -1 otherwise.

Defined in string.h

---

int strrpos (char *string, int c)

Returns the position of the last occurence of character c in the string, and -1 otherwise.

Defined in string.h

---

char *strrev(char *s)

Lattice compatible routine to reverse a string.

Defined in string.h

---

char *strset(char *s, int c)

Set all characters of a string to a specified value.

Defined in string.h

---

char * strrspn (const char string, const char * trimchars)

Return a pointer to the first character in 'string' to be trimmed (i.e. all characters from that point to the end of 'string' are in 'trimchars').

Defined in libgen.h

---

char *strrstr (const char * string1, const char * string2)

Return a pointer to the last occurrence of string2 within string1, or NULL if there is no occurrence. This is similar to tha ANSI compatible strstr() routine except that the search is done backwards from the end rather than forwards from the start.

Defined in string.h

---

char *strsrt(char *s, int c)

Lattice compatible routine for sorting an array of text pointers. However, tqsort() is more commonly used.

Defined in string.h

---

char * strtrns (const char * str, const char *old,
const char * new, char * result)

Transform 'str' and copy it into 'result'. Any character that appears in 'old' is replaced with the character in the same position in 'new'. A pointer to 'result' is returned.

Defined in libgen.h

---

char *strupr(char *s)

Convert a string to upper case.

Defined in string.h

---

void swmem(char *a, char *b, int n)

Defined in string.h

---

void sync (void)

Unix compatible routine to flush all memory buffers, and write the filing system Super Block. There is no direct equivalent in QDOS, so this is effectively a null call.

Defined in unistd.h

---

long tell(int fd)

Report position in a Level 1 file.

Macro defined in fcntl.h

---

long telldir(DIR *dirptr)

Unix compatible routine to report current position in a directory.

Defined in dirent.h

---

int toascii(int c)

Macro

Defined in ctype.h

---

int truncate (char * filename, off_t length)

Unix compatible routine to truncate a named file to a specified length.

Defined in unistd.h

---

void tqsort(char *ta[], int n)

Defined in stdlib.h

---

char * ttyname (int file_descriptor)

Get the name for the given file. On the QDOS implementation this will only work if the file was opened in this program using open() or fopen(). It will give an "<unknown>" reply for open files inherited from other programs. Returns NULL on error.

Defined in stdlib.h

---

void tzset(void)

Set the timezone dependent fields according to the setting of the TZ environment variable.

Defined in time.h

---

mode_t umask (mode_t)

Emulate the Unix/Posix system call for setting and reading the fie creation mask. In practise as QDOS and/or SMS do not support the concept of file permissions in the Unix/Posix sense this call is only a dummy and has no real affect.

Defined in sys/stat.h

---

int ungetch (char c)

Lattice compatible routine to 'unget' a character that has been obtained via getch() or getche(). Only a single level of pushback is supported. Returns value of character.

Defined in stdio.h

---

int unlink(char *name)

Defined in fcntl.h

---

int utime(char *filename, struct utimbuf *times)

Routine to emulate the UNIX utime() call. Sets a file modification time. If the second parameter is NULL, then the time is taken from the QL's real-time clock. If not it is taken from the structure 'utimbuf'. As QDOS only has a single time field, the larger of these two values is used.

Returns:

0 success

-1 failure (and sets errno)

Defined in utime.h

---

int wait(int *ret_code)

Routine to emulate the UNIX wait() call. Process stops until one of its child processes exits, or returns -1 if there are no active child processes. Suspended child processes are ignored. If a job has more than 255 children this call can fail badly. Returns process id of child that exited, plus exit code of terminated job if address is passed for it in ret_code (pass NULL if not wanted).

Defined in stdlib.h

---

long write(int fd, char *buf, long size)

Writes size bytes from array starting at buf to file fd. Size may be greater than 32K. Returns number of bytes written or -1 if error.

Defined in unistd.h

GLOBAL VARIABLES

---

extern char * environ[]

Holds an array of pointers to the environment strings for this program. Terminated by a NULL entry. It is automatically updated by the putenv() system call.

Defined in stdlib.h

---

extern char *sys_errlist[]

An array of pointers to the text corresponding to each (positive) error code.

Defined in errno.h

---

extern int sys_nerr

The number of error messages defined (i.e. the highest error code recognised).

Defined in errno.h

---

char * tempnam(const char * dir, const char * pfx)

Unix compatible routine to create a name for a temporary file. The 'dir' parameter specifies the directory to be used. If 'dir' is NULL or not a

suitable directory name then the TMPDIR environment variable will be checked. If that fails the P_tmpdir entry in the stdio.h header file is used.

Defined in stdio.h

---

extern long timezone

Defined in time.h

---

extern char tzdtn[4]

Defined in time.h

---

extern char tzstn[4]

Defined in time.h

---

extern char *tzname[2]

Defined in time.h

AMENDMENT HISTORY

25 Aug 93	The itoa() description changed to come in line with accepted usage (parameters reversed).
25 Jan 94	Added all the signal handling routines to the documentation. Some of these such as kill(), raise() and signal() had existed in embryonic form for some time, but the rest are new. Reworked the descriptions of the exec() and fork() calls to clear up some ambiguities.
20 Aug 94	Added description for chown(), link(), mknod(), rmdir() and umask() calls.
10 Oct 94	The documentation reworked to put the UNIX and POSIX compatible routines in their own file.
20 Oct 95	The documentation updated to reflect the implementation of Richard Zidlicky's signal handling extension.
07 Dec 96	Added descriptions for basename(), bgets(), copylist(), dirname(), strfind(), strrspn() and strtrns() routines.
12 Mar 98	Added description for strrstr() routine.

SCRPAR

NAME

scrpar_o V1.02 24/01/98

DESCRIPTION

"scrpar_o" is a C68 object code file which contains the following functions:

```
long scrb(long); /* returns base address of the screen */
long scrl(void); /* returns line length - in bytes */
long scrs(void); /* returns size of the screen - in bytes */
long scrx(void); /* returns X size of the screen - in pixels */
long scry(void); /* returns Y size of the screen - in pixels */
```

Please note that the above functions are likely to return the negative error codes associated with the following calls:

- ioa.open (IO_OPEN)
- ioa.clos (IO_CLOSE)
- iow.xtop (SD_EXTOP)

USE

These functions may be used in the same way as other "long" functions.

From V1.01 the function scrb() must be passed a QDOS channel ID as an argument. In this case, the returned value is the base address of the screen attached to the window corresponding to that channel ID. When the argument -1 is passed, the base address of the default screen is returned as usual.

COPYRIGHT

CHANGES

Release 1.00 (01/05/95)

First release. Only supports MODE 4 and MODE 8.

Release 1.01 (03/09/95)
A QDOS channel ID must be passed to scrb() as an argument (thanks are due to Christian van den Bosch without whom this release wouldn't have come into existence). This feature is especially intended for MINERVA users.

Release 1.02 (24/01/98)
This release has arisen from the need to maintain compatibility with the AURORA graphics card, resulting in a partial rewrite of "scrpar_s".

The improvements over previous releases are:

- scrs now returns the correct screen size under the AURORA card
- The scrl function has been added
- The code is more compact

Please note the following shortcoming:

- scrx now returns the same size in MODE 4 and in MODE 8

Command Index

COMMAND INDEX

The following is a short index in alphabetical order of the commands provided with the C68 system and the optional extra disks. In each case the disk on which it can be found is indicated in the square brackets.

ARC [C68 RUNTIME 2 before release 4.00]
[ARCHIVERS 1 disk]
Utility for maintaining an number of files in compressed archive format. The ZIP/UNZIP pair of programs are now more commonly used as they achieve better compression and are also more widely available on other operating systems.

AS68 [C68 RUNTIME 1]
The standard C68 assembler. Normally run via CC.

C68 [C68 RUNTIME 1]
The main C68 compile phase. Normally run via CC.

C68MENU [C68 RUNTME 2]
Front-end for running the C68 system completely from menus.

CAT [GNU TEXT UTILITIES]
Concatenate Text Files

CC [C68 RUNTIME 1]
The standard front-end for running the C68 compilation phases.

CfiX [CPORT SUPPORT LIBRARY]
A utility for taking the output from CPORT and completing the process of preparing it to be compiled by C68.

CMP [GNU DIFF UTILITIES]
Compare two text files

COMM [GNU TEXT UTILITIES]
Compare two sorted text files on a line-by-line basis.

CP [C68 RUNTIME 2]
A copy program. Normally used in conjunction with MAKE. Supprts recursive copying of directories.

CPP [C68 RUNTIME 2]
The C68 pre-processor phase. Normally run from CC.

CPROTO [C PROGRAMMING UTILITIES 1]
C prototype generator. Generates prototype definitions from C source files.

CSPLIT [GNU TEXT UTILITIES]
Split a text files into sections determined by context lines.

CUT [GNU TEXT UTILITIES]
Remove sections from lines of text files.

DIFF [GNU DIFF UTILITIES]
Finds the difference between two text files.

EXPAND [GNU TEXT UTILITIES]
Convert tabs in a text file to spaces

FGREP [C68 RUNTIME 2]
Finds which files (and lines) contain a specified string.

FOLD [GNU TEXT UTILITIES]
Wrap each line of a text file to fit within a specified width.

HEAD [GNU TEXT UTILITIES]

HEAD [GNU TEXT UTILITIES]
Output the first part of text files

INDENT [C PROGRAMMING UTILITIES 1]
Formats C source files to a specified standard layout.

INFOCMP [CURSES LIBRARY]
Display details from 'terminfo' database for the current terminal type.

JOIN [GNU TEXT UTILITIES]
Join lines of two text files on a common field.

LD [C68 RUNTIME 1]
The C68 linker. Normally run via CC.

MAKE [C68 RUNTIME 1]
The C68 utility for automating the compiling of programs.

NL [GNU TEXT UTILITIES]
Number lines of text files.

PACKHDR [C68 RUNTIME 2]
The C68 utility for packing C header files to occupy less room.

PASTE [GNU TEXT UTILITIES]
Merge lines of files

PR [C68 RUNTIME 2]
[GNU TEXT UTILITIES]
Convert text files for printing with page headings, line numbers,
columns etc.

RM [C68 RUNTIME 2]
Remove files. Normally used in conjunction with MAKE.

SED [C68 RUNTIME 2]
A stream editor. Normally used in conjunction with MAKE.

SLB [C68 RUNTIME 2]
The SROFF librarian. Used for manipulating libraries and analysing
SROFF files.

SORT [GNU TEXT UTILITIES]
Sort the lines of a text file.

SPLIT [GNU TEXT UTILITIES]
Split a file into pieces

SUM [GNU TEXT UTILITIES]
Checksum a file.

TAC [GNU TEXT UTILITIES]
Concatenate and print files with line order reversed.

TAIL [GNU TEXT UTILITIES]
Output the last part of files

TIC [CURSES LIBRARY]
Create a binary 'terminfo' database from a text version.

TOUCH [C68 RUNTIME 1]
Set file modification times. Normally used in conjunction with MAKE.

TR [GNU TEXT UTILITIES]
Translate or delete characters in a file.

TSORT [C68 RUNTIME 2]
Topological sort. Used in conjunction with SLB for ordering libraries.

UNEXPAND [GNU TEXT UTILITIES]
Convert spaces to tabs in a text file.

UNIQ [GNU TEXT UTILITIES]
Remove duplicate lines from a sorted file.

UNPROTO [C PROGRAMMING TOOLS 1]
Pre-processor for converting ANSI style function declarations to K&R
ones. Normally run via CC.

UNZIP [C68 RUNTIME 2]
[ZIP ARCHIVING UTILITIES]
Utility for extracting files from ZIP archives.

UUD [C68 RUNTIME 2]
Restore a file encoded with UUE. Documentation in file UUENCODE_DOC.

UUE [C68 RUNTIME 2]
Encode a binary file in ASCII text format for transmission over an
electronic mail network. Documentation in UUENCODE_DOC.

WC [GNU TEXT UTILITIES]
Print the number of bytes, words and lines in a text file.

ZIP [ZIP ARCHIVING UTILITIES]
Utility for maintaining a number of files in a compressed archive
format. This format is commonly used within the C68 system for source
files.

Table of Contents

Table of contents

- Overview
- Getting Started
- CC Front-End
- Gnu C Pre-Processor
- C68 Compiler
- AS68 Assembler
- Make
- C68 Linker
- C68 Menu System
- C68 environment on QDOS and SMS
- Environment Variables
- QDOS/SMS Signal Handling Extension
- QED Manual
- SROFF File Format
- Technical Reference
- TOS Emulator for QDOS
- Issue Notes for Release 4.22
- CP: Copy Files
- FGREP: Search Files
- GREP: Search for Patterns

Collected documentation for the C68 C Compilation System for the Sinclair QL.

" />

C68 Documentation

Dave Walker

Table of contents

next page