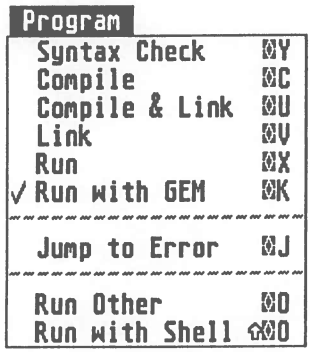
Compiling & Running Programs

The commands of this menu can only be used from LC.PRG with the exception of Run Other and Run with Shell.



Syntax Check

The Syntax Check item on this menu, checks the syntax of the source text that is currently being edited without producing any output file. Be sure to set up the INCLUDE environment variable so that the compiler can find the header files that you have #included. Because this command does not need to load the compiler or your program from disk, it lets you check your program quickly.

Compile

The Compile command is only available if you have loaded the second phase of the compiler LC2.TTP along with LC.PRG. Naturally this requires more memory than just loading the first phase of the compiler. This can be modified using the Preferences command.

Compile will produce a .O file that can be linked.If you haven't saved your program source code yet the file will be based on the name NONAME.

If you haven't loaded LC2.TTP the Compile command will be replaced by Thorough Check. This command only runs the first phase of the compiler but it will produce a .Q file that can be passed to LC2.TTP and is able to spot some errors that the Syntax Check command cannot.

Jump to Error

During a syntax check or compilation any warnings or errors that occur are remembered, and can be recalled from the editor. Clicking on Jump to Error from the Program menu, or pressing Alt-J will move the cursor to the next line in your program which has an error, and display the message in the status line of the window.

You can step to the next error by pressing Alt-J again, and so on, letting you correct errors quickly and easily. If there are no further errors when you select this option the message no more errors will appear, or if there are no errors at all the message What errors! will appear. Note that if there is more than one error on a line then only the first error is shown.

Note

If you are editing a file that is included by the current program, the version of the included file that is on disk will be used, so be sure to save any include file modifications to disk.

Link

If the compilation is successful you can then use the Link command to link your program. The linker will be run with a suitable command line. Obviously this requires the entire compiler as well as the editor and linker to be memory at once.

Compile & Link

You can combine the Compile and Link steps using the.Compile & Link command.

Run

If you have successfully compiled and linked a program you can then run it using the Run command. If your program crashes badly you may never return to the editor so, if in doubt, save your source code before using this, or the Run Other commands.

When issuing a Run command from the editor the machine may seem to 'hang up' and not run the program. This occurs if the mouse is in the menu bar area of the screen and can be corrected by moving the mouse. Similarly when a program has finished running, the machine may not return to the editor. Again, moving the mouse will cure the problem. This is due to a feature of GEM beyond our control.

Run with GEM

Normally, when the Run command is used, the screen is initialised to the usual GEM type, with a blank menu bar and patterned desktop. However if running a TOS program this can be changed to a blank screen with flashing cursor, by clicking on Run with GEM, or by pressing Alt-K. A check-mark next to the menu item means GEM mode, no check mark means TOS mode. The current setting of this option is remembered if you Save Preferences.

Note

Running a TOS program in GEM mode will look messy but will work, whereas running a GEM program in TOS mode can crash the machine.

Run Other...

This command is available from both versions of EdC. It lets you run other programs from within the editor, then return to it when they finish.

When you click on Run Other... from the Program menu you will first be warned if you have not saved your source code, then the GEM File Selector will appear, from which you should select the program you wish to run. If it is a .TOS or .TTP program you will be prompted for a command line, and then the screen will be initialised suitably.

This is the command to use for 'one-off' running of a program within the editor. If you are likely to want to run the same program a number of times, then use the facilities of the Tools menu. If you would prefer to specify the program to run via a command line, rather than using the File Selector then use the Run with Shell command described below.

If you include the character sequence %. (i.e. per cent followed by full stop) in the command line this will be replaced by the full name of the file that you are currently editing. To pass the name without its extension, use %?. Thus a command line of:

```
%?.o
```

would pass the name of the object file corresponding to the file being edited.

If you need a true % to be passed type %%.

Note

Screen initialisation depends on the filename extension, *not* the current Run with GEM option setting.

Run with Shell

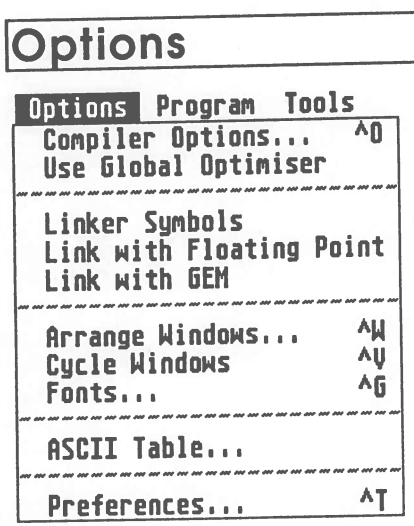
This command is available from both versions of EdC. It lets you run other programs from within the editor, then return to it when they finish. The keyboard shortcut for this command is Shift-Alt-O.

It differs from Run Other in that the you enter the file to run as a command line. If the editor finds that the _shell_p vector has been set up then this will be called to execute the command. This works well with the Craft shell as the shell can be used to run batch files and expand file wildcards etc.

If the _shell_p vector has not been set up then the editor will look for the file to run using the PATH environment variable.

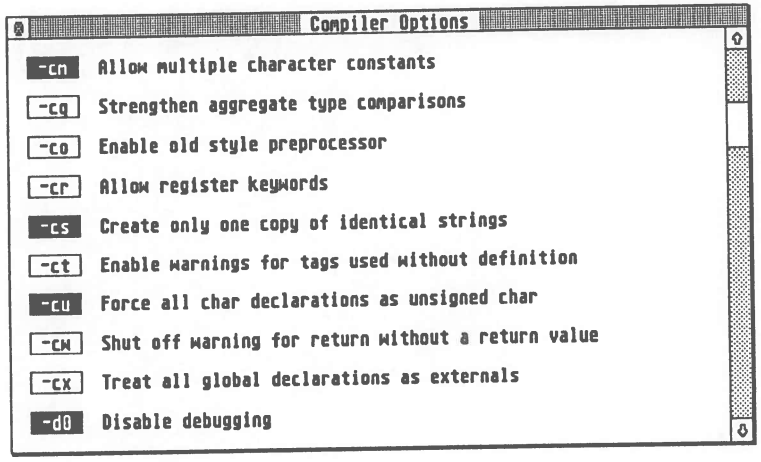
The same expansion of the current filename as used by Run Other can be used by this command. If you wish to use the same command more than once you will probably save time by using the Tools menu.

Options



Compiler Options

This command displays a large dialog box complete with scroll bar, like that shown below. It enables you to set all the compiler options that will be used when using the Syntax Check and Compile commands. To view further options click on the grey area to either side of the scroll bar to move a screenfull at a time or the arrows to move one line at a time.



Use Global Optimiser

If this option is checked (ticked) then the global optimiser will automatically be run when using the Compile command. This requires a lot of memory so that it is normally only possible to invoke it interactively on a system with more than one megabyte of memory.

Linker Symbols

If this option is selected (shown by a check mark) then the Link command will generate symbols in the executable file, ready for use by the MonST2C debugger.

Link with Floating Point

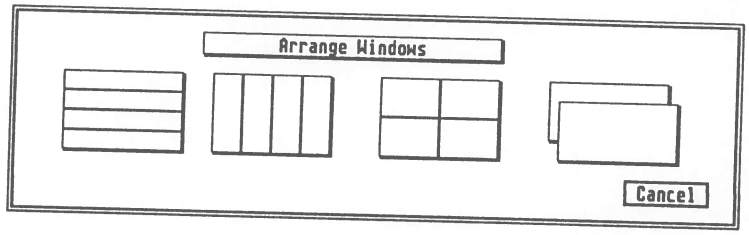
You should select this option if you are using the linker interactively and your program uses floating point as it causes your program to be linked with the floating point maths library.

Link with GEM

You should select this option if you are using the Link command and your program uses GEM as it causes your program to be linked with the GEM library.

Arrange Windows...

This command is used to change how multiple windows are displayed on the screen. It can be selected either by clicking on Arrange Windows... from the Options menu, or by pressing Ctrl-G; just click on the appropriate icon and the windows will be re-arranged for you.



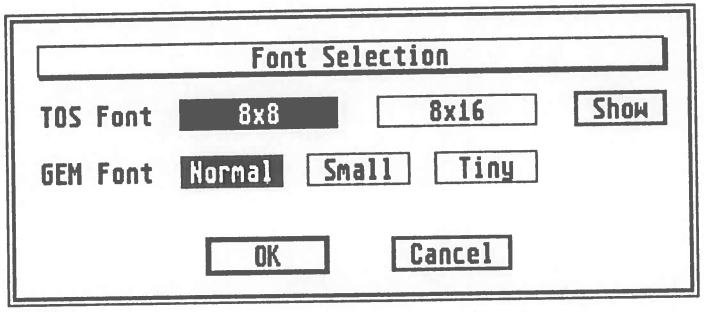
Clicking on the Cancel button will leave the windows arranged as they were before the Arrange Windows command was invoked.

Cycle Windows

This command is used to cycle between the active windows; i.e. if two windows are open it will swap between them at each usage. If three are open it will select first 1, then 2, then 3 and then 1 again.

Fonts...

The Fonts command is used to select different GEM or TOS fonts, it can be selected either by clicking on Fonts... from the Options menu, or by pressing Ctrl-W. It displays a dialog box like this:



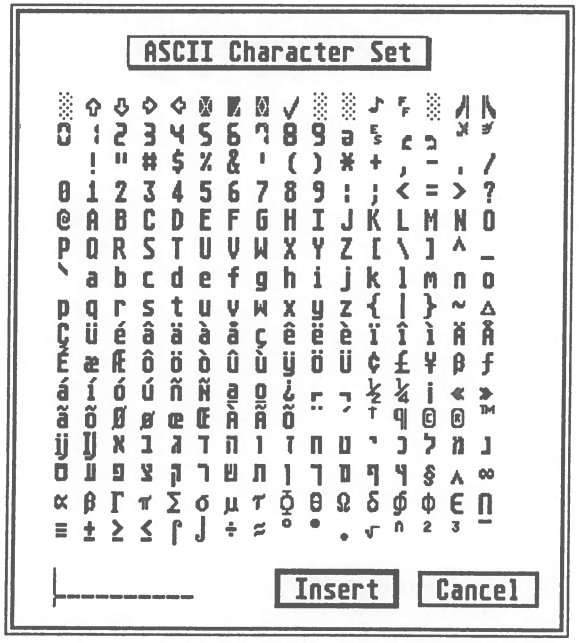
The GEM Font is the font that will be used by the editor to display text. In monochrome there are three fonts available as above. Changing to Small will double the number of line displayed on the screen to 40. With the Tiny font the characters are only 6 pixels by 6 pixels wide but this does mean that there are over 100 characters per line and 54 lines!

In medium resolution, there are only two fonts; Normal and Small. Small is 6 by 6 pixels and thus the characters are difficult to read but this does give an extra 7 lines of text and over 100 characters per line.

TOS font is used by non-GEM programs such as the compiler. If you click on the Show button then a sample piece of text is printed using the TOS font so that you can decide whether or not the selected font is legible on your monitor. On standard monochrome monitors using 8x8 will give 50 lines instead of 25; in medium resolution using 8x16 gives only 12 lines.

ASCII Table...

This command displays a dialog box like that below, showing all the ASCII characters:

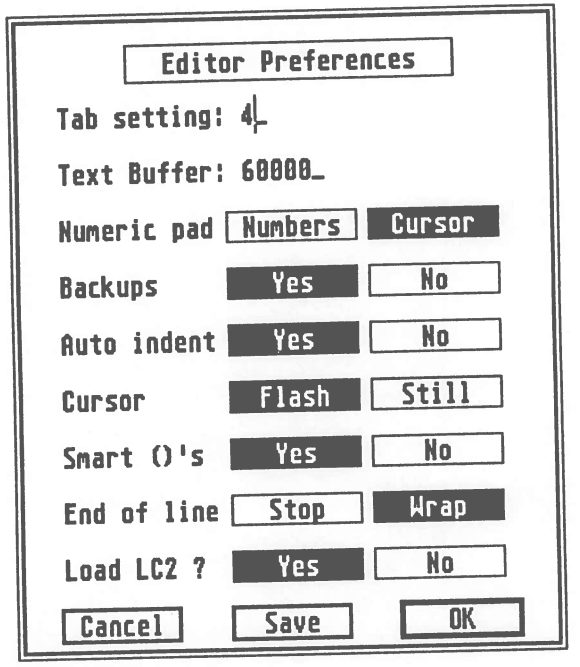


You may click on individual characters and they will be shown in the line at the bottom of the box. Pressing Return or clicking on the Insert button will then add the characters to the text that you are editing at the current cursor position.

Note that the characters that would confuse the editor are 'greyed out' and may not be selected. Remember when using characters other than the standard 7 bit ASCII ones that these symbols are not the same on other computers.

Preferences...

Selecting Preferences... from the Options menu will produce a dialog box like this:



Tabs

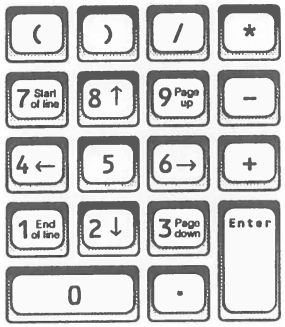
By default, the tab setting is 4, but this may be changed to any value from 2 to 16.

Text Buffer

By default the text buffer size is 30000 bytes, but this can be changed from 4000 to 999000 bytes. This determines the largest file size that can be loaded and edited. This amount of memory is allocated for each window in use. You can see how much free memory you have by pressing the Help key. Changing the editor workspace size will cause any text you are currently editing to be lost, so a confirmation is required if it has not all been saved.

Numeric pad

The Numeric pad option allows the use of the numeric keypad in an IBM-PC-like way allowing single key presses for cursor functions, and defaults to Cursor pad mode. The keypad works as shown in the diagram below:



This feature can be disabled, if desired, by clicking on the Numbers button.

Backups

By default the editor does not make backups of programs when you save them, but this can be turned on by clicking on the Yes button.

Auto indenting

It can be particularly useful when editing programs to indent subsequent lines from the left, so the editor supports an auto-indent mode. When active, an indent is added to the start of each new line created when you press Return. The contents of the indent of the new line is taken from the white space (i.e. tabs and/or spaces) at the start of the previous line.

Cursor

By default the EdC cursor flashes but this can be disabled if required.

Smart ()s

This facility lets you check that your parentheses match. When you press) the cursor will quickly move to any matching (character and then back to the current position, thus you can ensure that you have closed the correct number of brackets in a complex expression. If you find this cursor movement distracting then disable it.

End of Line

By default (Stop), when you press cursor left at the beginning of a line or cursor right at the end of line, the cursor does not move. Changing this item to Wrap causes the cursor to move to the previous line if you press cursor left at the beginning, and to the next line if you press cursor right at the end.

The best way to find out which you prefer is to try using each setting.

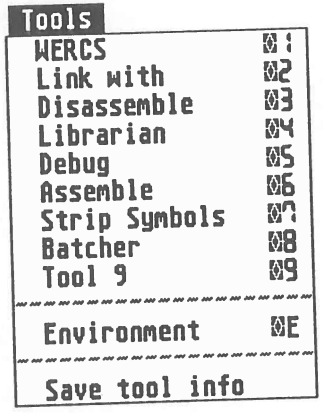
Load LC2

This option determines whether LC.PRG will load the second phase of the compiler when it loads. The new value of this option will only have an effect if you save the preferences and re-execute LC.PRG.

Saving Preferences

If you click on the Cancel button any changes you make will be ignored. If you click on the OK button the changes specified will remain in force until you quit the editor. If you would like the configuration made permanent then click on the Save button, which will create the file EDC.INF on your disk. Next time you run EDC.PRG or LC.PRG the configuration will be read from that file.

Tools Menu

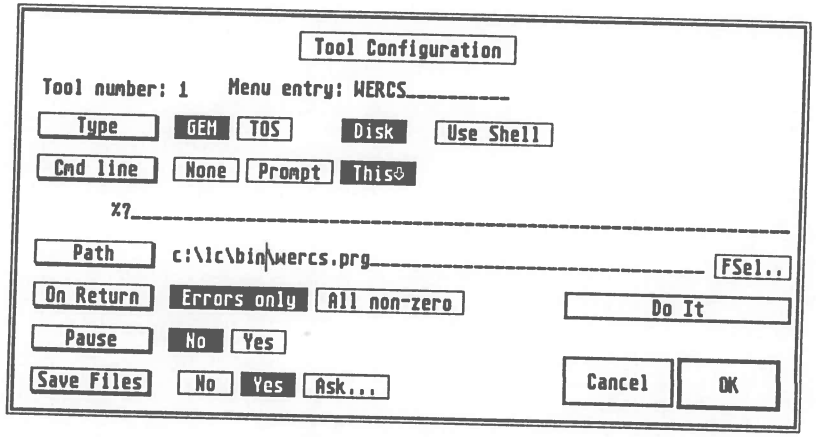


The Tools menu lets you run programs of your choice from within the editor using a single keystroke or click of the mouse. To take full advantage of this facility you will need at least one megabyte of RAM and either a hard disk or two double-sided floppies.

The configuration can be saved in a EDCTOOLS.INF file, ensuring that the same facilities can be used again, the next time that you run the editor.

The EDCTOOLS.INF file that we supply is set up to run many of the tools supplied with Lattice C.

Before you can use this facility you will need to configure each tool so that the editor can find the appropriate file. To configure a tool, hold down the Ctrl key and select the appropriate menu item or press Ctrl Alt and the appropriate key *on the numeric keypad*. This will produce a dialog box like this:



If you just want to use the default settings, you need only change the Path item so that the file can be found; either amend this item or click on FSel and use the file selector to select the appropriate file. Once you have made the required changes you should press Return (or click on OK) to make your changes permanent; alternatively pressing Cancel will ignore any changes you have made. The other options in this box are:

Menu entry

The name typed in this field gives the name of the tool as placed on the Tools menu. Hence in the above example the name WERCS appears on the menu.

Type

The button selected here changes how the program is run, either as a GEM program or as a TOS program; note that the same warnings about GEM/TOS mode made under Run with GEM apply here also.

Disk/Use Shell

These buttons control which of the two run commands is actually used: Run Other or Run with Shell. If Disk is selected then the Path specified must be a complete path, otherwise simply a name will do for the Use Shell option.

Cmd line

These options configure the way the command line is obtained for a program which is about to be run. If None is selected then a program will be run as a plain GEM or TOS program with no command line. If Prompt has been selected you will be prompted for a command line in the same way as occurs when using Run Other.

Finally This↓ allows the command line on the line below to be used. This command line is specified in the same way as that used by Run with Shell and may have the same meta-characters in it.

On Return

This option allows you to specify which errors EdC will bring to your attention when returning. If Errors only is selected then you will only be alerted to negative return codes from programs, i.e. those normally indicating GEMDOS errors. All non-zero will also force positive program error returns to be flagged.

Pause

This option controls whether the editor pauses after running the tool. Typically you will select Yes when running a TOS program and No when running a GEM program.

Save Files

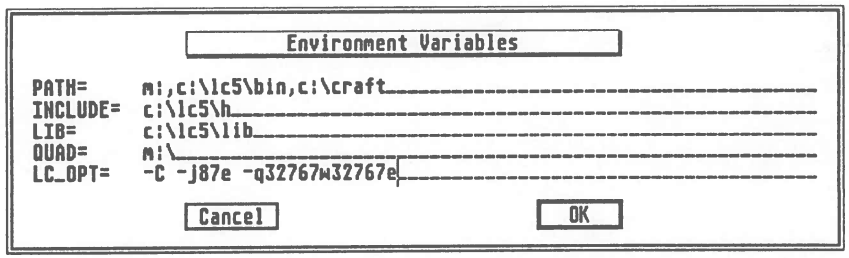
These options change which files will be saved before running the tool. If you select No then no files will be saved, selecting Yes (the default) will save *all* files (not just the current window), whilst Ask... will prompt you using the Save/Leave dialog described under Quitting EdC.

Running Tools

To run a configured tool is simple, just select the appropriate menu item or press Alt and the appropriate key *on the numeric keypad* and the program will be run using the settings described above.

Environment

The environment option allows the environment variables used by the tools which are run to be altered. Only the variables which are needed are shown:



The settings displayed may then be altered to reflect any changes you may wish to make. The environment variables used by the compiler are discussed in the section **LC, The Compiler**.

Save tool info

This command saves the current tool settings, environment variables and compiler options. It will create the file EDCTOOLS.INF on your disk. Next time you run EDC.PRG or LC.PRG the configuration will be read from that file.

Miscellaneous Commands

About EdC

It you click on About EdC... from the Desk menu, a dialog box will appear giving various details about EdC include the free memory left to the system. Pressing Return or clicking on OK will return you to the editor.

Help Screen

The key equivalents for the commands not found in menus can be seen by pressing the Help key, or Alt-H. A dialog box will appear showing the WordStar and function keys, as well as the free memory left for the system.

Switching Windows

EdC has support for up to four windows, which can be selected by pressing Alt-1 to Alt-4 (on the top row of numbers, *not* on the numeric pad). To load into a new window you should normally use the Load Another...command (Ctrl-L) described earlier. You can also switch windows by clicking on the appropriate window with the mouse.

To cut and paste between windows is just as simple as copying blocks in a single window, i.e. mark the block and then use Remember Block command, switch windows (as described above) and then Paste Block.

Windows & Desk Accessories

The Editor Windows

The windows used by the editor work like all other GEM windows, so you can move them around by using the move bar on the top of it, you can change their size by dragging on the size box, or make them full size (and back again) by clicking on the full box. Clicking on the close box will close the current window. If you close the last window EdC will ask you if you want to quit or have a new untitled window.

Desk Accessories

If your ST system has any desk accessories, you will find them in the Desk menu. If they use their own window, as Control Panel does, you will find that you can control which window is at the front by clicking on the one you require. For example, if you have selected the Control Panel it will appear in the middle of the screen, on top of the editor window. You can then move it around and if you wish it to lie 'behind' the editor window, you can do it by clicking on the editor window, which brings it to the front, then re-sizing it so you can see some part of the control panel's window behind it. When you want to bring that to the front just click on it and the editor window will go behind. The editor's cursor only flashes and the menus only work when an editor window is at the front.

Automatic Double-Clicking

You may configure EdC (or LC) to be loaded automatically whenever a source file is double-clicked from the Desktop, using the Install Application option.

To do this, go to the Desktop, and click once on EDC.PRG (or LC.PRG) to highlight it. Next click on Install Application from the Options menu and a dialog box will appear. You should set the Document Type to be C, and leave the GEM radio button selected. Finally click on the OK button.

To test the installation, double-click on a file with the chosen extension (which on old, 1.0, ROM machines must be on the same disk and in the same folder as EdC) and the Desktop will load EdC, which will in turn load the file of your choice ready for editing.

Note To make the configuration permanent, you have to use the Save Desktop option.

Saved! Desk Accessory Users

If you use the PATH feature of the Saved! desk accessory then the restriction of having your data files in the same folder and drive as EdC described above is not relevant. The editor looks for the EDC.INF configuration file firstly in the current directory (which is the folder where you double-clicked on the data file), then using the system path. Saving the editor preferences will put the .INF file in the same place it was loaded from, or if it was not found then it will be put in the current directory.

You may invoke Saved! from within the editor at any time by pressing Shift-Clr. This will only work if the desk accessory is called SAVED!.ACC or SAVED.ACC on your boot disk.

LC
The Compiler

The Lattice C Compiler can be run either using the integrated compiler described in the section **EdC, The Screen Editor**, or from the command line. This first section below describes running from a command line interpreter (such as Batcher or Craft). The subsequent two sections are relevant to users of both the integrated and CLI environment, describing the environment variables and compiler options.

The Compiler Driver

Command line operation of the Lattice C Compiler is invoked via the lc.ttp command. The lc program separates the options list into those for pass 1 and those for pass 2. Options to the compiler are specified as a list of minus (-) prefixed letters placed *before* the file names; any options after the first file name will be ignored.

LC1 (pass 1) and LC2 (pass 2) are then executed for each of the C source files specified by the files list, with the optional, third, global optimisation phase between pass 1 and pass 2. The file name list can consist of one or more file names and/or file patterns, separated by white space. For example:

```
lc * \mydir\myprog \mydir\abc?
```

will compile all C source files in the current directory, plus the source file named \mydir\myprog.c plus all C source programs in the \mydir directory which have four-character names beginning with abc. Note that the lc command automatically supplies the .C extension on all source file names.

The lc command will also automatically invoke the librarian and linker if required.

Return Codes

The lc command returns the following completion codes:

0 All compilations were successful. That is, at least one source program was compiled, and there were no fatal errors.

1 One or more fatal compilation errors were reported.

2 No source files were found.

The Compiler Phases

The compiler is normally split into two phases (with an optional third, global optimisation, phase). These two phases are known as lc1 and lc2; note that they are not normally called explicitly, but instead via the compiler driver lc.ttp or the integrated environment lc.prg.

The Parser and Pre-processor

The lc1 and lc1b commands invoke the first compiler pass, which reads a source file and translates it into an intermediate form known as a *quad* file.

lc1b invokes the big compiler for cross referencing and prototyping purposes. this will be automatically invoked if the -g option or the prototyping options are used on the lc command. Note that the compiler included in the integrated compiler, LC.PRG, is a big compiler hence all prototyping and listing options may be used within it.

Unlike the lc command, you can only specify one source file to lc1, and it should be written without the .C extension. For example, if the file argument is myprog, this pass will translate myprog.c into the quad file named myprog.q.

The options can consist of the following items, which are described above:

-b	Base register relative data addressing.
-c	Compiler compatibility settings.
-d	Debugging mode or preprocessor symbol definition.
-e	Extended character set processing.
-f	Floating point format selection.
-g	Listing generation options. This is only valid with lc1b.
-h	Precompiled header file inclusion.
-i	Directory paths for local include files.
-j	Error/warning message control.
-l	Longword alignment of data items.

-n	Retain only eight characters in symbol names.
-o	Specify destination for .Q file. Note that this is specified as -q on lc.
-p	Preprocessor options. This is only valid with the lc1b command.
-q	Compilation error abort control.
-r	Subroutine call control.
-u	Undefine preprocessor symbols.
-w	Generate code to use short integers.
-x	Treat all global declarations as externals.

The code generator

lc2 reads a quad file and translates it into an object file. The options can consist of the following items, which are described above:

-m	Select target architecture.
-o	Specify destination for .O file.
-s	Specify segment name.
-v	Disable stack checking.
-y	Unconditionally load the base register.

The global optimiser

The global optimiser, GO, analyses a quad file, performs several types of optimisations, and produces another quad file. This type of transformation makes the use of the optimiser completely optional since its input file is the same format as its output file. In many cases, optimised code is more difficult to debug than non-optimised code so frequently the optimiser is only used after the main program has been tested and is mostly working.

Since the optimiser works on quads, the Lattice machine independent intermediate form, it has no knowledge of the target processor or its instructions. The code generator contains all of this knowledge and makes very full usage of the 680x0 instruction set. The code generator tries not to generate extra instructions in the first place but it does have a peep-hole optimiser to catch the few places where this is not possible. The following optimisations are performed:

Register assignment

Commonly used auto, formal, and temporary variables are assigned to registers for all or part of their lifetime, according to usage.

Dead store elimination

Stores of values which are never fetched again are eliminated.

Dead code elimination

Code whose value is not used is eliminated.

Global common sub-expression merging

Recalculation of values that have been previously computed is eliminated. GO performs this with function scope.

Hoisting of invariants out of loops

Calculations performed inside a loop whose value is the same on each iteration of the loop are moved outside the loop.

Induction variable transformations

Loops containing multiplications, usually associated with indexing, have the operations reduced in strength to addition.

Copy propagation

Definitions of the form leftvar = rightvar are eliminated when all uses of leftvar have this definition as the single reaching definition, and the variable rightvar will not change before each use. This optimisation primarily exists to support other optimisations.

Constant propagation and folding

References to variables whose only definition is a constant are replaced by the constant. Often the definition is eliminated if all references are replaced. GO performs constant folding to propagate the new constants further.

Auto variable elimination and re-mapping

Unused auto variables are eliminated, and storage offsets are reassigned. Often the variable is unused because of previous optimisations.

Very busy expression hoisting

Code size is reduced by moving an expression computed along all paths from a point in the code to a common location. For instance, in

```
if (a())
  f(i + j);
else
  g(i + j);
```

the expression I+J will be computed in only one place.

Various reductions in strength

GO will perform associative re-ordering of additive operations involving constants, to reduce the operation count.

Various arithmetic operations involving constants are reduced in strength.

Conditional and logical expressions whose result is unused are converted into corresponding If() code. For instance, putchar(_) from <stdio.h> is implemented with a conditional expression. If the result (the original character or an error indication) is not used, GO converts it into if-else code, eliminating a load into a register.

Various control flow transformations

GO will perform various transformations to eliminate unreachable code or useless control structures.

Reordering to reduce value lifetimes

Expressions with a single use are moved adjacent to the operation that uses them. This helps reduce temporary lifetimes, and supports optimisations that move code around. For example, in

```
p[i] = f(_);
```

the computation of the address &p[i] can be moved after the call.

Environment Variables

The compiler uses the environment variable feature of GEMDOS to locate the various programs and files. Such assignments allow these programs and files to be located in any directory on any disk.

Setting the variables

Environment variables may be set in one of a number of ways. If running from within the integrated environment, they are normally manipulated using the Environment command described in the section **EdC, The Screen Editor**; if running from Batcher (or another shell, e.g. Craft) they are set as described therein.

An additional compiler option is available for users who are running from the Desktop (or from a shell which does not support environment variables), -E, which is followed by an environment variable and value, e.g.

```
-EPATH=c:\bin
```

The environment variables recognised and used by the compiler are:

PATH — Executable path

This variable defines where the driver will look when trying to locate the different programs which it needs to invoke (i.e. LC1, LC2, GO etc.). It should consist of a list of semi-colon (;) or comma (,) separated items, for instance:

```
PATH=c:\lc5;c:\bin
```

would search the directory c:\lc5 first followed by c:\bin. Note that the current directory is always searched first.

INCLUDE — Include path

The INCLUDE variable is similar to the PATH variable except that it is used to locate the include files used by your program, so that:

```
INCLUDE=c:\lc5\h;c:\myhdrs
```

would search the directory c:\lc5\h first followed by c:\myhdrs.

LIB — Library path

The LIB environment variable instructs the linker where the library files may be found, so that:

```
INCLUDE=c:\lc5\lib;c:\mylibs
```

would search the directory c:\lc5\lib first followed by c:\mylibs.

Note that just because a library file is in the library directory does not mean that the file will be linked in, you must tell the compiler this using the -L+ option.

QUAD — Quad path

The QUAD environment variable specifies the default intermediate (QUAD) file name used by the compiler. If the filename has a trailing backslash (\) then the compiler assumes that this is the name of a directory such that it may form a filename by concatenating the source file name to it.

If you have a RAM disk installed you can greatly increase compiler performance if you use this as the quad temporary directory. If your RAM disk was drive M then you would use the assignment:

```
QUAD=m:\
```

Alternatively if you wished to place this files on your hard disk in a folder called quads, this can be done as:

```
QUAD=g:\quads\
```

LC_OPT — Default options

This variable gives the default compiler options. When the lc driver starts it reads this variable and inserts the options at the start of the command line, so that you can include your favourite options automatically. For example if you always want continuous compilation and any number of errors or warnings you might set LC_OPT to:

```
LC_OPT=-C -q-
```

Pre-processor Symbols

During pre-processing, the compiler defines several symbols prior to (and during) compilation so that you may investigate the translation environment. The following symbols are defined at the start of all compilations:

Name	Value	Meaning
__DATE__	"date"	Date on which compilation was started
__FILE__	"name"	Name of main file which is being compiled
__LINE__	n	Current line which is being translated
__REVISION__	6	Current minor version number.
__STDC__	0	ANSI operation mode
__TIME__	"time"	Time at which compilation was started
__VERSION__	5	Current major version number.
ATARI	1	Host Machine
LATTICE	1	Compiler Name
LATTICE_50	1	Compiler Version
LATTICE_56	1	Current compiler release
M68000	1	Processor type

The following symbols may also be defined depending on the current compiler options:

Name	Option
_ANSI	-ca
_BASEREL	-b1
_DEBUG	-d1..-d5

LPTR	without -w
_M881	-f8
_MDOUBL	-fd
_MLATTICE	-fl
_MMIXED	-fm
_MSINGLE	-fs
__PLAIN_CHAR_UNSIGNED	-cu
_PCREL	-r1
_REGARGS	-rr
_SHORTINT	-w
SPTR	-w
_UNSIGNEDCHAR	-cu

Note that any of the non __ prefixed symbols may be undefined via -uXX.

Compiler Options

The compiler options below all apply to the command line driver, lc.ttp; options which are not available in the integrated environment are noted. The list to lc.ttp can contain any combination of the following, separated by blanks:

-B This option causes the lc command to always use the lc1b compiler rather than lc1. This is useful if you have enough ram to run the big compiler but wish to economise on disk space. This option is ignored by the integrated compiler.

-b This option causes the compiler to change the form of addressing used to locate statics, externals and strings. By default, -b1 is used to imply that all such items are addressed as a 16 bit offset from address register A4. The disadvantage of this is that it only allows 64K bytes of data to be addressed. You can override this option by using the -b0 option which implies full 32 bit addressing for accessing all items.

Note that this option does not limit the amount of data that may be allocated at run time using malloc.

This option is passed to lc1 where it actually causes the compiler to change the default storage class of statics to far or near as appropriate. If you have a program which has a large amount of data, you can readily use the -b1 default by putting the far keyword on any large objects to move them out of this common merged data section.

-C This option causes the lc command to continue with the next source file when a fatal compilation error is reported while multiple source files are being compiled. Normally, a fatal error causes the process to pause with the following message displayed on your screen:

```
Compiler return code xx.
Press Y to abort, any other key to continue.
```

The compiler error messages are also displayed immediately above this prompt. If you respond with a Y (yes), lc will abort, otherwise it will proceed to the next source file. This option is ignored by the integrated compiler.

-c The compiler defaults to compatibility with previous releases with many ANSI C language features, but the -c *compatibility option* can be used to activate some important features as well as compatibility with other compilers. The -c must be immediately followed by one or more letters from the following list, in any order. We recommend that you use the options -cusf for the best code generation and error reporting.

Note that all -c options are toggle options, i.e. specifying any such option twice will disable it.

+ Compatibility mode for the Lattice C++ product. This will suppress warnings associated with structure passing and other potential problems areas that will have already been diagnosed by the C++ front end.

a Enables full ANSI compatibility mode with full diagnostics to check for portability problems. Some features of the compiler are disabled when this option is specified, such as precompiled header files and suppressing multiple includes of the same file in order to achieve compliance. It also disables register (-cr) and extra (-ck) keywords, also the warning messages 122 - "Missing ellipsis", 132 - "Extra tokens after valid preprocessor directive" and 135 - "Assignment to shorter data type (precision may be lost)" are enabled.

c Allows comments to be nested.

d Allows $ character to be used in identifiers.

e Suppresses the printing of the error source line in conjunction with any warnings or errors.

f Forces the compiler to check for the presence of function prototypes and to complain when one isn't present at a function call or function definition.

i Suppresses multiple #includes of the same file. If a second #include of the same file is encountered, the directive is simply ignored. Note that case is important although no distinction is made for angle brackets or quotes. This option is implied when precompiled header files are used or created.

k Enables the presence of the near and far keywords even when the -ca option has been specified.

l This forces alignment of all external data to longword boundaries. Note that this option is far more useful than the apparently similar option -l, which forces alignment of all objects (including structure members) resulting in structures which are potentially incompatible with TOS.

m Allows use of multiple character constants (e.g. 'ab').

o Provides a compatibility mode to use the pre-ANSI style preprocessor found in previous releases of the compiler. The most important aspect of this occurs in substitution of symbols within quoted strings.

q strengthen the aggregate equivalence type checker. When disabled (the default), this option allows two aggregates with common initial subsequences over the length of one of the aggregates to type check equivalent.

r Enables the register keywords __d0 to __a7, even when the -ca option has been specified.

s Causes the compiler to generate a single copy of all identical string constants into the code section of the program. Note that when this option is specified, modification of any string constants at runtime will produce unpredictable results.

t Enables warning messages for structure and union tags that are used without being defined. For example:

```
struct XYZ *p;
```

would not normally produce a warning message if structure tag XYZ was not defined.

u Forces all char declarations to be treated as unsigned char.

w Shuts off warning messages generated for return statements which do not specify a return value within an int function. For conformance with the ANSI standard, all such functions should be declared as void instead of int.

x Causes all global data declarations to be treated as externals. This is identical to specifying the -x option.

-d This option has two uses. When used by itself or immediately followed by a numeric digit, it activates the debugging mode. Currently, the following debugging options are supported:

-d0 Disables all debugging information.

-d1 Enables output of the line number/offset table.

-d Same as -d1.

-d2 Outputs full debugging information for only those symbols and structures referenced by the program.

-d3 Outputs full debugging information for only those symbols and structures referenced by the program. Additionally it will cause the code generator to flush all registers at line boundaries.

-d4 Outputs full debugging information for all symbols and structures declared in the program even if there is no reference to them.

-d5 Outputs full debugging information for all symbols and structures declared in the program even if there is no reference to them. Additionally it will cause the code generator to flush all registers at line boundaries.

When any of the debugging options is specified, the preprocessor symbol DEBUG will be defined so any debugging statements in the source file will be compiled.

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

-dsymbol
Causes symbol to be defined as if your source file had the statement:

```
#define symbol
```

-dsymbol=value
Causes symbol to be defined as if your source file had the statement:

```
#define symbol value
```

-e This option causes the compiler to recognise the extended character set used in Asian-language applications.

-Esymbol=value
Causes symbol to be defined in the environment with the given value. This can be used to set up environment variables for the compiler outside of the integrated environment or shell (e.g. to set the PATH variable). This option is ignored by the integrated compiler.

-f This option controls the format to be used for all floating point operations. Currently two basic styles of floating point are supported:

-f8 Inline Motorola 68881 generated instructions using the co-processor interface. Code compiled with this option will not operate unless a 68881 is installed which conforms to this interface. Note that the linker will also demand the 68881 specific library routines; these are only available (at present) as part of Lattice C/TT.

-fa Auto-detecting I/O based 68881 emulation routines will be used when this option is specified. The library will check for the presence of an I/O based 68881 (such as Atari's SFP004) and perform floating point arithmetic on chip.

-fi I/O based 68881 maths routines will be used when this option is specified. The library assumes the presence of an I/O based 68881 (such as Atari's SFP004) and performs floating point arithmetic on chip.

-fl Standard Lattice IEEE routines linked into the program to perform software emulation of all floating point operations. This code will work on all machines but will not take advantage of a 68881 if present. This option is the default for compatibility with previous versions of the compiler.

In addition to the floating point styles, the compiler allows some control over the precision attributed to the float and double declarations used within the user code. If you specify both a floating point style and a precision, it must be done on the same -f option such as in -flm or -f8s.

-fs Causes the compiler to treat all declarations as single precision.

-fd Causes the compiler to treat all declarations as double precision.

-fm Causes the compiler to treat float as single precision and double as double precision. This option is the default for all formats.

-f Will reset to the default of Lattice IEEE mixed mode.

-g This option causes the big version of lc1 to generate a cross reference and listing file. lc will automatically invoke this version of the compiler if the option is specified. The -g option is followed by one or more of the following option letters in any order:

c Outputs a cross reference of all compiler-provided include files found by searching the directories specified by the INCLUDE environment variable. By default these symbols are not printed.

d Includes all #define symbols in the cross reference listing.

e Causes the source listing to display all excluded lines as controlled by #if or #ifdef. Normally these lines are not displayed.

h Includes the contents of all include files found in the default include directory as they were included by the source program. Normally, only the #include directive causing the compiler to read the file is displayed.

i Includes the contents of all user-provided include files in the expanded listing.

m Displays both the original source line and the line after macro expansion in the listing. This is useful for tracking down problems related to preprocessor replacement of symbols.

n Toggles the narrow mode of the listing. By default, the listing will be formatted for a 108 column line with most lines not exceeding 80 characters. When enabled, this option allows for listing lines up to 132 characters.

s Toggles listing of the input source code.

x Toggles generation of a cross reference of the symbols encountered in the source file.

-H This option specifies that the compiler is to preload the symbol table from a precompiled header file. It is immediately followed by the name of the precompiled header file as in:

```
-Hinclude\all.sym -Hall.sym
```

There is no limit to the number of precompiled header files that may be read in.

-i This option specifies a directory that the compiler should search when it is attempting to find an include file. For example, if you specify the option as -Ia:\headers -Ib:\local and then place the line:

```
#include "defs.h"
```

in your source program, the compiler first tries to find the header file named defs.h in the current directory. If it is not there, then the compiler searches for a:\headers\defs.h and b:\local\defs.h in this order. Finally, if these attempts fail, the compiler will attempt to open the file from the places specified in the INCLUDE environment variable.

Note that you can use up to 16 -I options.

-j This option allows control over the error messages reported by the compiler. It is immediately followed by a number and then an optional letter:

-jn Causes the compiler to suppress printing of warning number n.

-jne Causes the compiler to treat any occurrences of warning n as an error instead.

-jni Causes the compiler to suppress printing of warning number n.

-jnw Enables printing of warning n. By default, several ANSI oriented messages are disabled.

Several messages may be affected with the same -j option such as -j22l30e132w which disables warning message 22, turns 30 into an error and enables 132 as a warning.

-L When this option is present, lc invokes the linker if all compilations are successful. The first source file name is used as the name of the executable and map files produced by the linker. Any other files that were compiled are supplied to the linker as secondary object files. The Lattice C startup routine is included as the first object module, with an appropriate standard library file (lc.lib) searched last.

Additional Lattice libraries and linker options may be specified by immediately following the -L option with one or more of the following letters:

a This invokes the XADDSYM option of the linker. It causes HiSoft extended debugging information for all routines to be output in the executable file.

b This invokes the BATCH option of the linker. It forces batch mode linking.

f This invokes the MAP option of the linker. It causes a map file to be generated with the .MAP file extension.

g This letter specifies that the GEM AES and VDI library lcg.lib is to be searched before the standard run-time support library. When this option is specified the default extension for the output file becomes .PRG rather than .TTP.

h This letter directs the linker to output the hunk portion of the map. This is the default map if no other map options are specified.

l This letter directs the linker to include library information in the map file.

m This letter specifies that the Lattice IEEE maths library lcm.lib is to be searched before the standard run-time support library.

n This invokes the NODEBUG option of the linker. It causes all debugging information to be stripped from the final executable.

q This invokes the QUIET option of the linker. It causes no messages to be output by the linker if a link is successful.

s This letter directs the linker to produce a symbol listing in the map file.

v This invokes the VERBOSE option of the linker. It causes the linker to display statistical messages as it is processing the object files and libraries.

x This directs the linker to include cross reference information in the map file.

For example, -Lm will search lcm.lib before lc.lib, and -Lvg will search lcg.lib and lc.lib, and display messages regarding the current linker status. Note that the standard libraries are always searched last.

If you want to search other libraries, you must list those libraries after the option letters, and use plus signs as separators. For example, -L+myfuncs.lib searches myfuncs.lib before the standard Lattice library, while -Lm+myfuncs.lib+\george\myfuncs.lib searches the libraries myfuncs.lib, \george\myfuncs.lib, lcm.lib and lc.lib. Note that the special libraries are searched before the Lattice libraries.

The -L option creates a file in the current directory named xxx.lnk, where xxx is the name of the first source file to be compiled (i.e., the same name that is used for the executable and map files). This .LNK file serves as input to the linker, and it is not deleted at the end of the procedure. This allows you to easily re-link if, during your testing, you find a need to change and re-compile only one module. To do this, simply execute CLink in the following way:

```
clink WITH xxx.lnk
```

where xxx.lnk is the name of the .LNK file previously produced by the lc command.

-l This option causes all objects except characters, short integers, and structures that contain only characters and short integers to be aligned on longword boundaries (i.e. addresses exactly divisible by 4). Structures will be longword aligned if they contain any members that must be aligned. This option can be used on full 32 bit machines (e.g. the Atari TT) to increase performance by reducing the need for half-word memory accesses.

-M When this option is present, lc will compile only those source files with dates more recent than the corresponding object files. Note that the dates of included files are not checked. In other words, if you change a header file without changing the source file that includes it, the source file will not automatically re-compile because it still pre-dates its object file.

For larger projects where there is a more intense dependency upon structures in common data files being changed, we recommend using a make utility to manage recompilation of the affected source files automatically. This option is ignored by the integrated compiler.

-m This option allows control of the type of code generated. The -m must be immediately followed by one or more letters from the following list in any order:

0 Causes the compiler to generate code which will run on a Motorola 68000. Decisions on code optimisation will be based on the timings for this processor.

1 Causes the compiler to generate code which will run on a Motorola 68010. Decisions on code optimisation will be based on the timings for this processor. In general, code for this will run on a 68000 although the 68010 has instructions not found on the 68000.

2 Causes the compiler to generate code optimised for the 68020 processor. This code will not run on a 68010 or 68000 although it will run on a 68030.

3 Causes the compiler to generate code optimised for the 68030 processor. This code will not run on a 68010 or 68000 although it will work on a 68020.

a Causes the compiler to generate code to run on any Motorola 680x0 family processor. Code is optimised for the 68020/68030, degrading performance on a 68000.

c Disables the deferred stack cleanup optimisation which leaves parameters on the stack, after a call, to be reused and cleaned up by a subsequent subroutine call or function epilogue.

r Disables the automatic registerisation of variables. By default, the compiler will attempt to pick likely candidates for register variables.

s Causes the compiler to choose optimisations which result in a reduction of space instead of time.

t Causes the compiler to choose optimisations which result in a performance increase at the cost of code space. This is the default.

-n This option causes the compiler to retain only 8 characters for all identifiers. The default maximum identifier length is 31 characters. In either case, anything beyond the maximum length is ignored. Note that this option is the reverse of that in the version 3 release of the Lattice C.

-O This option invokes the global optimiser. This option is ignored by the integrated compiler.

-o This option should be followed by the drive, directory, or complete file name for the object file that is produced by pass 2. Several examples are:

-oa:
Places the object file in the root directory on drive a:.

-o\obj
Places the object file into directory \obj\ on the current drive. The name of the file is the same as the source file name, with a .O extension instead of .C.

-ospecial.o
Places the object file into the current directory with the name special.o.

-p This option is used when using the compiler in a preprocessor mode to produce a file used by subsequent compiler invocations. When this option is used, the compiler will not create a quad file. However, the file specified as the -o option will be used as the target name for the created file. There are several uses for the -p option:

-p By itself, -p, causes the compiler to write the results of preprocessing the input source file into the output file. If no output file is specified, a file extension of .p will be used to create the file.

-ph Causes the compiler to generate a precompiled header file containing a dump of all symbols encountered in the given source file. This file may then be used for the -H option on subsequent compiler invocations to reduce compilation time.

-pr Causes the compiler to generate a prototype file containing prototypes for all functions defined in the source file. The -pr may be immediately followed by one or more of the following option letters in any order:

- **e** Eliminates prototypes for all static functions. Only those functions available externally will have prototypes generated for them.
- **p** Causes the compiler to generate prototypes with __PROTO for portability to other compilers.
- **s** Generates prototypes for all static functions. Only those functions defined with the static function will be output.

Note that -pres will not generate any prototypes.

-q This option has two uses. If the -q is immediately followed by a letter, it specifies where the quad file is to be generated. Otherwise it is used to control how many errors/warnings will be allowed before quitting a compilation.

This option should be followed by the drive, directory, or complete file name for the quad file, which is the intermediate file generated by pass 1 and read by pass 2. Several examples are:

-qm:
Places the quad file in the root directory of drive m:.

-q\quad
Places the quad file into directory \quad\ on the current drive. The name of the file is the same as the source file name, with a .Q extension instead of .C.

Note that the quad file is automatically deleted by pass 2.

To control the maximum number of errors/warnings, the -q should be immediately followed by a number then either an e or w. For example:

-q3w
Quit after 3 warnings or errors.

-q2e
Quit after 2 errors.

-q10w1e
Quit after 10 warnings or any errors.

-q
Quit after any errors or warnings.

-q-
Never quit on any errors or warnings.

Note that when the compiler quits due to too many errors/warnings, it will not generate a quad file.

-r This option is used to control how the compiler is to generate subroutine calls and entries. The -r option may be followed by one or more of the following characters in any order:

- **0** Defaults all subroutine calls to far which means that the compiler will use an absolute 32-bit relocated address to locate the target function. Note that any functions explicitly declared near will use the more efficient 16-bit relative offset.
- **1** The compiler default, causes all subroutine calls to be defaulted to near which means that the compiler will use a 16-bit PC relative address to locate the target function. In order for this to work, the target subroutine must be within +/-32K of the generated instruction. If it is not within range, the linker will generate an ALV to allow the call to be bridged to the final target. Any functions explicitly declared far will use the larger 32-bit address.
- **r** Causes the compiler to use registerised parameters for all subroutine calls and entry points. The first two integral and two pointer items will be loaded into d0-d1/a0-a1 for the call. Any function without a prototype or explicitly declared __stdargs will use the normal stack conventions.
- **s** The compiler default, causes the compiler to use standard stack parameters for all subroutine calls. Those functions explicitly declared __regargs will use registerised parameter conventions.
- **b** Defaults the compiler to use registerised parameters for all subroutine calls, yet still generate a prologue that handles both styles of parameter passing.

-R When this option is specified, the object modules produced by the compiler are automatically inserted into a library file, replacing modules of the same names. The option must be followed by the name of the library, as in

```
-Rmylib.lib
```

which places the object modules into the mylib.lib library file. The -R can be followed by any valid file name, including drive code and path. A .lib extension is not automatically supplied. This option is ignored by the integrated compiler.

-s This causes the compiler to use the default names of text for the program section, data for the data section, and udata for the bss or uninitialised data section.

-sc=codename
Causes the compiler to use the name codename for the program, or code, section without affecting the names of the other sections.

-sd=dataname
Causes the compiler to use the name dataname for the data section without affecting the names of the other sections.

-sb=bssname
Causes the compiler to use the name bssname for the bss, or uninitialised data, section without affecting the names of the other sections.

-t This option is used to change the initial startup code linked when using the -L option. The -t option should be followed by one of the following characters:

a This option forces the use of the desk accessory startup code when linking. It also has the effect of changing the default extension on the final output file to .ACC.

d This option forces the use of the automatic program type detection code. The external variable _XMODE can be used to determine the current mode.

r This option forces the use of the resident program startup code. The use of this startup type is rather specialised and is discussed in the linker section.

=file This allows the specification of an alternate startup code. The file argument should consist of a complete pathname specifying the location of the required startup code.

Note that more information of the various startup stubs is provided in **Appendix F - The Lattice C Start-Up.**

-u This option by itself undefines all preprocessor symbols which are normally pre-defined by the compiler. The -u option may be followed by a name causing that name to be undefined:

-uNAME
Causes NAME to removed from the predefined pre-processor word set.

-v Disable the generation of stack checking code at the beginning of each function.

-w This option causes the compiler to treat all integers as 16-bit short values. It is intended to provide compatibility with other compilers although it does provide an increase in performance of the generated code. When using this option, we strongly recommend use of prototypes to catch parameter mismatch errors as not all parameters will be promoted to 4 bytes, as is the default.

-x Cause all global data declarations to be treated as externals. This can be useful if you define data in a header file that is included by multiple source files. The -x option can be used with all the files except one, in this case, to cause the data items to be defined in one module and referenced as externals in the others.

-y This option causes each function entry sequence to load address register A4 with the value of the linker defined symbol _LinkerDB. This symbol is the data section base address, biased as necessary. This option must be used if the -b1 option is used with interrupt code. Note that, in general, only the functions that can be used as entry points to the interrupt handler need to use this feature, since register A4 will be propagated by subsequent function calls. -y is superseded by the __saveds option keyword that may be used with a function. Any function having this keyword will automatically load up the base register upon entry.

-z Cause the compiler to generate GST format linkable code. It is intended to provide compatibility with other languages, and is not recommended due to the poor performance of linkers using this format, their inability to generate ALVs for out of range branches and the lack of general support for base-relative addressing via A4.

Pre-compiled Header Files

Pre-compiled header files provide a method for speeding up compilation of programs which have large numbers of static include files (i.e. do not interact dynamically). Say, all modules of your program have the following statements:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <aes.h>
#include <vdi.h>
#include "mystruct.h"
#include "globals.h"
#include "depend.h"
...
```

These header files may be pre-compiled by building a 'dummy' file which simply includes the above files. This file is then compiled using the -ph option in addition to your normal compiler options; note that this will produce a 'quad' file in the normal location, hence typically the object file is explicitly specified via -q, e.g.

```
lc -ph -qinclude.sym include.c
```

On subsequent compilations of the main file the pre-compiled file is pre-loaded using the -H option:

```
lc -Hinclude.sym myfile.c
```

Note that there is no need to remove the includes from the file being compiled as the use of pre-compiled headers implies the -cI option.

Language Extensions

Lattice C 5 adds several new keywords to the C language, some of these are specified by the ANSI C standard, whilst others are extensions added to support easier or better access to special facilities of the compiler.

The extensions to the ANSI standard are preceded by a double underscore, such as __near, as is required by the standard. If the -ca flag has *not* been specified then the compiler also accepts the extended keywords in the more natural form without the double underscore prefix.

ANSI Extensions

const

The const type is used to declare an initialised data item that will never change. For example

```
char const name[] = "12345abc";
```

declares a constant string. This modifier is also often used in a function prototype where a pointer is passed. Using this modifier can help the code generator since it may be able to extend the lifetime of an object over a function call.

enum

The enum type is used to declare an integral item that can only have certain named values, each of which is treated as an integral constant. The actual values assigned to the identifiers normally begin at zero and are incremented by one for each successive identifier. An explicit value can be forced by using an equals sign, then subsequent identifiers are assigned the new value plus one, etc.

For example, this statement defines an enum type:

```
enum colour {red, blue, green=4, puce, lavender};
```

and this defines some objects of that type:

```
enum colour x, *px;
```

In this example, the symbols associated with the enumerated type colour are given the following values:

Value	Name
0	red
1	blue
4	green
5	puce
6	lavender

Each enumeration is a separate type with its own set of named values. The properties of an enum type are identical to those of the int type.

signed

The signed keyword is treated exactly like the unsigned keyword and ensures that a particular variable will be treated as signed. In practice this is only useful with character types when using the the -cu option to force characters to be treated as unsigned.

void

The void type indicates the empty type, and can be used in several ways; when used as a function return value or as a cast, it indicates that the value is to be discarded, e.g.

```
void john(int x);

(void)printf("Hello World\n");
```

void may also be used to indicate a function which takes no parameters:

```
void fn(void);
```

Note that this is *not* equivalent to the declaration void fn() which indicates nothing about the parameters, in particular it *does not* mean that no parameters are used.

The final usage introduced by ANSI was the generic pointer, void *. This is used in a similar way to the way older code used char * as a generic pointer, so that a generalised pointer may be manipulated without knowing what it points to. Because void is the empty type de-referencing void * is illegal, i.e.

```
void *p;

if (*p)
  ...
```

will generate error 29, invalid pointer operation.

volatile

The volatile keyword describes a data object that can be changed by means outside the control of the declaring program. Examples of such objects are memory-mapped I/O registers and shared memory. When manipulating a volatile object, the compiler reads or writes the object whenever it is referenced. In other words, the compiler suppresses any optimisations that would keep volatile objects in registers.

Storage Classes

Several keywords are provided to indicate the storage class of an object. With previous versions of the compiler, the only way to change the storage class was to use the -b option. This option is still available, but the recommended method is to let the compiler default to near addressing, -b1, and then use the keywords where necessary. Unlike MS-DOS based compilers, these keywords do not affect the size of an object, but instead indicate the storage class. In that vein, you must place the keyword as close to the data item as possible:

```
int near x; /* addressed a4 relative */
long far y; /* addressed with 32 bit absolute */
```

Note that you can only use the storage keyword immediately before the target object.

far

The far type indicates that the object must be accessed with a full 32-bit address rather than via a 16 bit base-relative pointer.

huge

The huge keyword is identical to far when using Lattice C on the ST. It is included for compatibility with other environments which use Intel processors instead of the Motorola 68000 family.

near

The compiler uses the near access method for objects declared using the near keyword. For example,

```
int near x, near y, near z;
```

declares three near integers. These are placed into the data section in such a way that they can be accessed via 16-bit offsets from the data section pointer in register A4. The -b1 option on the lc command causes all data declarations without a specific access method to be treated as near. This is the default setting for the -b option. In other words, the compiler normally generates near objects in order to reduce program size and improve performance.

Note that because of storage class model used, declaration of pointers using near and far is slightly unusual; consider the definitions:

```
int *near x;  /* define near pointer to object */
int near *x;  /* define pointer to near object */
```

because of the storage class model, the first definition causes the pointer to be in the near data section, whilst the latter definition has no effect on the code generated since it indicates a pointer to near data (which is is not relevant to the 680x0 code model).

Notice that pointers to near objects are always 32 bits wide. The only time that the 16-bit access occurs is when the offset can be embedded within an instruction. For most near objects this is frequently the case, and so the size and performance improvements can be substantial. However, if you normally address an object via a pointer, you will gain little by declaring that object as near.

Calling Conventions

The Lattice C 5 compiler also provides a number of keywords that may be applied to functions to permit special calling conventions. The __regargs, __stdargs and __asm keywords indicate that the compiler is to use an altered calling convention.

The default is to use __stdargs for all functions. However, if you use the -rr option of the compiler then it will use the __regargs convention in which the first two data items and first two pointer items are passed in d0/d1 and a0/a1, respectively. The keywords allow you to override the default. For example:

```
long __regargs foo(int i) { ... }
void __stdargs bar(void);
```


Note that the keywords __asm, __stdargs, and __regargs are mutually exclusive. Full details on using these keywords is given in the section **ASM, The Assembler**.

__asm

The __asm keyword allows you to specify, exactly, in which register each parameter is to be passed. It can be used for both function definitions and function declarations:

```
int __asm mymax(register __d0 int,register __d1 int);

int __asm myfun(i,p)
  register __d0 int i;
  register __a1 char *p;
```

__interrupt

The __interrupt keyword is applied to a function to indicate that this function may be called from an interrupt routine. Although, at the time of writing, it does not affect the code generated for the function, it is provided for potential variations in code necessary to support interrupts.

__regargs

This keyword defines a subroutine that is to be called with register parameters. Note that full function prototyping *must* be used so that the compiler can decide which parameters are of which type.

__saveds

If a function may be called from code which has not set up the global base register (A4) then it is necessary to load it at the start of the function. This is possible using the -y option. However this applies to all functions in a module; to cause it to be loaded for a single function, you can use the keyword __saveds as in:

```
int __saveds myentry(void)
{
  ....
}
```

Note that __saveds only has meaning when applied to the actual definition of the function. External functions with the __saveds keyword simply ignore the keyword.

__stdargs

This keyword defines a subroutine that is to be called with standard stack parameters.

Built-in Functions

The Lattice C 5 compiler provides several standard library functions which are built-in to the compiler and as such generate high quality 680x0 machine code exploiting register contents in a way which would not otherwise be possible. Since the compiler 'knows' the semantics of these functions it may pre-compute constant expressions, like strlen("Hello World"); also if a function result is not used, it may be discarded before it is computed.

The built-in functions recognised by the compiler are all prefaced with __builtin_ and then followed by their standard library name. The header files use #defines to ensure that the built-in function is used instead of the library version, for example:

```
int strlen(const char *);
int __builtin_strlen(const char *);

#define strlen(a) __builtin_strlen(a)
```

Such a mechanism ensures that it is possible to suppress the use of a built-in function and force the library definition to be used. This can be useful if you wish to have, for instance, the mem family of functions check their input parameters against the bounds of your heap, i.e. to catch dangling or random pointers.

To force the use of the library version you should include the normal header files and then explicitly #undef the function, e.g.

```
#undef strlen
```

The library functions recognised by the compiler as built-ins are:

```
int abs(int);
int max(int,int);
int memcmp(const void *,const void *,size_t);
void *memcpy(void *,const void *,size_t);
void *memset(void *,int,size_t);
int min(int,int);
int strcmp(const char *,const char *);
char *strcpy(char *,const char *);
size_t strlen(const char *);
```


In addition to these, the printf function, in its __builtin_printf form, is recognised. When such a call occurs the formatting string is analysed according to the normal library rules to see if it contains:

- No substitutions, so that a call to _writes may be made,
- Only %d, %p, %s and %x conversions, when a substitution is made for _tinyprintf.

Otherwise a call to the standard library printf routine is made.

The compiler also makes available several built-in functions which increase the functionality of the language:

```
void __emit(short);
void __builtin_fpc(int,double);
void geta4(void);
long getreg(int);
void putreg(int,long);
```

Again these are normally prefixed by __builtin_, with the following suffices being acted upon:

__emit	This function inserts its short word argument into the instruction stream at the current point. This can be used to insert unusual instructions into the program, for instance:
	`__emit(0x27c); /*and #$dfff,sr*/`
	`__emit(0xdfff);`

..._fpc	__builtin_fpc is used to generate inline MC68881 transcendental instructions using the Line-F opcodes. It takes two parameters, the second of which is the operand to be passed to the function for evaluation, whilst the first is the 'encoded extension field', i.e. the low 7 bits of the FPC opcode. Consider the inlining of the function sin:
	`double sin(double);`
	`#define sin(x) __builtin_fpc(14,x)`

geta4	This 'function call' forces the global data register, A4, to be loaded at the start of a function. It is exactly equivalent to using the __saveds keyword on the function definition, but may be used in portable code with a placebo definition of geta4 being used in a non-Lattice environment.

getreg	getreg directly obtains the contents of a specific register; this can be useful in situations where you need to pick up specific register values, e.g. the stack pointer.
putreg	putreg allows you to store a value into a specific register.

Using these functions allows direct access to the instruction stream and code generation. Note that, whilst code may be inserted in the instruction stream using __emit, it is often easier and more useful to use the #pragma inline capability of Lattice C 5 described below.

Inline Calls

The #pragma inline directive allows the Lattice C compiler to generate inline code, either to support direct calling of the operating system or to use features of the processor not supported by C.

The directive has the form:

```
#pragma inline [<r>=] <name> ([<parms>])
  { [register <s1> [, <s2>] [,...]] ["<emit>";[...;]]}
```

The various parts of the directive are:

<r>	the register in which the function returns its value.
<name>	the name of the previously prototyped function which is to be inlined.
<s1>,<s2> etc.	the registers which are destroyed as a result of this call.
"<emit>"	the hexadecimal string to be placed in the instruction stream.

<parms> gives the manner in which the parameters are passed to the call as follows:

```
([<cast> | <r1>] [, [ <cast> | <r2> ] ] [,...])
```


where:

<cast>	an optional cast to (short) so that the parameter is placed on the stack as a short (rather than the natural size for the type).
<r1>	a register in which the parameter is to be passed.

The | separators above indicate alternatives, so that a parameter may be cast *or* assigned to a register. The [...] notation indicates that the enclosed parameters are optional, so that a <parms> value may even consist of commas with no intervening casts or register assignments.

Consider calling the GEMDOS function Cconout, this takes a single parameter which is the character to be printed. Before executing the GEMDOS trap, we must also specify GEMDOS function number, 2 in this case:

```
#define __TRAP_1    "4e41"

void _vgs(short,short);

#define Cconout(c) _vgs(2,c)

#pragma inline _vgs((short),(short))
  { register d2,a2; __TRAP_1;}
```

This results in the parameter c being pushed onto the stack as a short-word followed by the function number 2 as a short word, followed by the GEMDOS trap. Prior to the call the registers D2 and A2 are saved.

Calls to more complex parts of TOS may also be effected; consider the linead function:

```
#define __LINEA_D   "a00d"

void linead(int,int,LA_SPRITE *,void *);

#pragma inline linead=(d0,d1,a0,a2)
  { register d2,a6; __LINE_A_D;}
```

This calls the Line-A sprite routine with the co-ordinates in D0 and D1 and the sprite definition and save blocks in A0 and A2. Prior to the call, D2 and A6 are saved. Note that A2 is not saved, along with D2 and A6, as this is implicit in its usage in the call.