

Lattice C 5.5

*the C Compiler for your Atari
ST/STE/TT Computer*

Addendum

User Manual

Requires:

- ✓ Atari 520ST upwards
(1M+ memory required)
- ✓ Disk drive
(2 floppies or hard disk advised)
- ✓ Mouse

HiSoft
High Quality Software

Lattice C 5.5 for the Atari ST/STE/TT

By HiSoft and Lattice, Inc.

© Copyright 1992 HiSoft and Lattice, Inc. All rights reserved.

Program:

designed and programmed by HiSoft and Lattice, Inc.

Manual:

written by Alex Kiernan and David Nutkins.

This guide and the Lattice C program diskettes contain proprietary information which is protected by copyright. No part of the software or the documentation may be reproduced, transcribed, stored in a retrieval system, translated into any language or transmitted in any form without express prior written consent of the publisher and copyright holder(s).

HiSoft shall not be liable for errors contained in the software or the documentation or for incidental or consequential damages in connection with the furnishing, performance or use of the software or the documentation.

HiSoft reserves the right to revise the software and/or the documentation from time to time and to make changes in the content thereof without the obligation to notify any person of such changes.



Published by HiSoft

The Old School, Greenfield, Bedford MK45 5DE UK

First Edition, March 1992 - ISBN 0 948517 56 5

Table of Contents

Lattice C 5.5	1
<i>A word about pop-up menus and dialogs</i>	2
<i>The Editor's windows</i>	5
<i>Switching Windows</i>	6
<i>Entering text and moving the cursor</i>	7
<i>Cursor keys</i>	7
<i>Tab key</i>	8
<i>Backspace key</i>	8
<i>Delete key</i>	8
<i>The Edit menu</i>	9
<i>Go to top of file</i>	9
<i>Go to end of file</i>	9
<i>Goto line</i>	9
<i>Block Commands</i>	10
<i>Marking a block</i>	10
<i>The Clipboard: Copy, Cut & Paste</i>	11
<i>Saving a block</i>	11
<i>Copying a block</i>	11
<i>Deleting a block</i>	12
<i>Copy block to block buffer</i>	12
<i>Pasting a block</i>	12
<i>Printing a block</i>	12

Deleting text	13
Delete line	13
Delete to end of line	13
UnDelete Line	13
Delete block	13
Searching and Replacing Text	14
Bookmarks	15
Disk Operations	16
New	16
Loading Text	16
Revert	17
Save As...	17
Save	17
Inserting Text	17
Delete File	18
Close	18
Change Directory	18
Quitting Lattice C	19
Configuring the editor	20
Auto-indent lines	20
Auto-save configuration	20
Auto-save project	21
Cursor Mode Numeric pad	21
Hide mouse when typing	21
Make Backups	21
Show matching parentheses	22
Stop at End of Line	22
Save files on Quit	22

Save files on run other	22
Tab setting	22
Text Buffer	23
Cursor	23
Load...	23
Saving preferences	23
Reset	24
Running other programs	24
Tools Menu	25
Menu entry	26
Command line	26
Directory	26
Save Files	27
Path	27
Pause on return	27
Report all errors	27
Run as TOS & Run as GEM	27
Make Resident	28
Running Tools	28
Run Other...	28
Run with Shell...	29
Setting the Path	29
Miscellaneous Commands	31
Fonts...	31
ASCII Table...	32
About Lattice C	32
Help Screen	33
Desk Accessories	33

Automatic Launching	33
Compiling Programs	34
The Project menu	34
New...	35
Load...	35
Save "..."	35
Save As ...	35
Edit "..."...	36
Make "..."	37
Make all "..."	38
Link "..."	38
Run "..."	38
Debug "..."	38
Directory	38
Problems	39
Compilation Errors	39
The Program menu	40
Assemble	40
Check	40
Compile	40
Pre-compile	41
Pre-process	41
Prototype...	41
Generate <code>__PROTO</code> style prototypes <code>-pp</code>	42
No identifiers in prototypes <code>-pi</code>	42
No typedefs in prototypes <code>-pt</code>	42
Functions	42

The Options menu		43
Environment...		43
Lattice C and the environment		43
Executable path	PATH	44
Include path	INCLUDE	44
Library path	LIB	44
Quad file	QUAD	45
General...		45
Compiler options		46
Compiler options - Advanced		47
Allow nested comments	-cc	47
Allow \$ in identifiers	-cd	47
Allow explicitly-sized bitfields	-cb	47
Disable trigraph processing	-cg	47
Enable 'near'/'far' keywords	-ck	47
Enable '__asm' keywords	-cr	47
Make 'extern' declarations global	-cx	48
Make external definitions 'extern'	-x	48
Make string literals non-'const'	-ch	48
Type based struct equivalence	-cq	48
Source character set	-e	48
Float/double	-f	49
Pre-processor expansion buffer: size	-zsize	50
Precompiled headers	-Hfile.sym	50
Identifier significance: sig	-nsig	50
Compiler options - Errors		51
Disable 'return' warnings	-cw	51
Disable all warnings	-j*i	51
Enable all warnings	-j*w	51

<i>Make all warnings errors</i>	<i>-j*e</i>	51
<i>No error line printing</i>	<i>-ce</i>	51
<i>No error/warning limit</i>	<i>-q-</i>	51
<i>Stop on first error/warning</i>	<i>-q</i>	52
<i>Undefined tag warnings</i>	<i>-ct</i>	52
<i>Error limit: num</i>	<i>-qnume</i>	52
<i>Warning limit: num</i>	<i>-qnumw</i>	52
<i>Warnings:</i>		52
Compiler options - General		52
<i>Allow multi-character constants</i>	<i>-cm</i>	53
<i>Default short integers</i>	<i>-w</i>	53
<i>Disable stack checking</i>	<i>-v</i>	53
<i>Enforce function prototypes</i>	<i>-cf</i>	53
<i>Make 'char' unsigned</i>	<i>-cu</i>	53
<i>Merge identical strings</i>	<i>-cs</i>	53
<i>Strict ANSI mode</i>	<i>-ca</i>	53
<i>Suppress multiple includes</i>	<i>-ci</i>	55
<i>Processor</i>	<i>-m</i>	55
<i>Debug</i>	<i>-d</i>	56
<i>Maths</i>	<i>-f</i>	57
<i>Parameters</i>	<i>-r</i>	58
Compiler options - Listing		59
<i>Cross-reference source symbols</i>	<i>-gx</i>	59
<i>Cross-reference system includes</i>	<i>-gc</i>	59
<i>Cross-reference user includes</i>	<i>-gd</i>	59
<i>Display error messages</i>	<i>-go</i>	59
<i>List source code</i>	<i>-gs</i>	59
<i>List system includes</i>	<i>-gh</i>	60
<i>List user includes</i>	<i>-gi</i>	60

Narrow listing	-gn	60
Show #if/#ifdef excluded lines	-ge	60
Show macro expansions	-gm	60
Compiler options - Object		61
Always generate stack frame	-mf	61
Auto-load base register	-y	61
Default 'far' code	-r0	61
Default 'far' data	-b0	62
Default section names	-s	62
Disable auto-registerisation	-mr	62
Disable stack merging	-mc	62
Long align externals	-cl	62
Long align stack	-as	62
Optimise for space	-ms	62
Type based stack alignment	-aw	63
Code: codename	-sc=codename	63
Data: dataname	-sd=dataname	63
BSS: bssname	-sb=bssname	63
Data pointer: reg	-breg	63
Frame pointer: reg	-rreg	63
Register limits - Data: num	-hdnum	64
Register limits - Address: num	-hanum	64
Register limits - Floating point: num	-hfnum	64
Compiler options - Optimiser		64
Assume best case aliasing	-Oalias	64
Disable register colouring	-Onocolor	65
Enable global optimisation	-O	65
Enable loop invariant hoisting	-Oloop	65
Optimise for	-O	65

Compiler options - Pre-processor		66
Allow #if to span files	-cp	66
Allow nested #define's	-cn	66
Old-style pre-processor	-co	66
Undefine all symbols	-u	66
#undef' symbols: name	-uname	67
#define' symbols:		67
#include' directories: dir	-idir	67
Assembler options		67
Add '_' to symbol names	-u	68
Add line number information	-d	68
Allow multiple listing lines	-lm	68
Allow 68851 instructions	-m9	68
Allow 68881 instructions	-m8	68
List source file	-l	68
List include files	-li	68
List macro expansion text	-lx	68
Processor	-m	69
Identifier significance: sig	-nsig	69
EQU' symbols:		69
INCLUDE' directories: dir	-idir	70
Executable options		70
Build GEM application	-lg	70
Clear GEMDOS "Fastload" bit	NOFASTLOAD	70
Load program in TT RAM	TTLOAD	70
Perform "Malloc"s from TT RAM	TTMALLOC	70
Standard symbol format	DRISYM	71
TT RAM TPA size: size	TPASIZE size	71
Application type		71

<i>PREFIX' file: file</i>	<i>PREFIX file</i>	72
Linker options		72
<i>Add exported symbols</i>	<i>ADDSYM</i>	72
<i>Ignore errors</i>	<i>IGNORE</i>	73
<i>Ignore symbol casing</i>	<i>NOCASE</i>	73
<i>Strip debugging information</i>	<i>NODEBUG</i>	73
<i>Linker buffer size: size</i>	<i>BUFSIZE size</i>	73
<i>Messages</i>		73
<i>ALVs</i>		73
<i>DEFINE symbols:</i>		74
Map...		75
<i>Cross reference external symbols</i>	<i>MAP...X</i>	75
<i>List external symbols</i>	<i>MAP...S</i>	75
<i>Map input file placements</i>	<i>MAP...F</i>	75
<i>Map input section placements</i>	<i>MAP...H</i>	75
<i>Map library file placements</i>	<i>MAP...L</i>	75
<i>File name width: width</i>	<i>FWIDTH</i>	75
<i>Page height: height</i>	<i>HEIGHT</i>	75
<i>Hunk name width: width</i>	<i>HWIDTH</i>	75
<i>Line indentation: indent</i>	<i>INDENT</i>	75
<i>Program name width: width</i>	<i>PWIDTH</i>	75
<i>Symbol name width: width</i>	<i>SWIDTH</i>	75
<i>Form width: width</i>	<i>WIDTH</i>	75
Librarian options		76
<i>Cross-reference symbols</i>	<i>-x</i>	76
<i>Generate symbol listing</i>	<i>-s</i>	76
<i>Verbose operation</i>	<i>-v</i>	76

Debugger options	76
Auto '@'/'_' prefix labels	77
Auto-load source	77
Display 'ZAn' in disassembly	77
Enable timed screen switching	77
Follow TRAPs	77
Ignore cartridge area	78
Ignore label case	78
Interpret relative offsets	78
Symbol significance: sig	78
Source line numbers	78
Resident configuration	79

LC1, LC2, GO **81**

New Language Features	81
ANSI compliance	81
Extern scoping model	81
Flexible keyword ordering	82
Float as single	82
Redundant keyword combinations	82
Ref/def model	82
Restriction of register arrays/aggregates	83
Scoping rules for 'no-linkage' objects	83
Trigraphs	83
Type composition of scoped declarations	83
typedef model	83
Valid storage classes of local functions	83
C++ features	84
Comments	84

<i>Ellipsis</i>	84
<i>Anonymous unions</i>	84
<i>Floating point __asm support</i>	84
<i>__interrupt keyword</i>	85
<i>ANSI relaxations</i>	85
<i>Modifiable lvalues</i>	85
<i>Signed and sized bit fields</i>	85
<i>Zero length arrays</i>	86
<i>Listing control directives</i>	86
<i>#pragma eject</i>	86
<i>#pragma space lines</i>	86
<i>#pragma title title</i>	86
<i>Error control directives</i>	87
<i>#pragma error num</i>	87
<i>#pragma ignore num</i>	87
<i>#pragma warning num</i>	87
<i>New error/warning messages</i>	87
<i>Pre-processor symbols</i>	90
<i>Optional definitions</i>	90
<i>Static definitions</i>	91
<i>Permanent definitions</i>	91
<i>Changes to the run-time model</i>	91
<i>Register passing mode</i>	92
<i>__saveds and stack checks</i>	92
<i>A2 as a register variable</i>	92
<i>__asm functions</i>	92
<i>Signed and sized bit fields</i>	93

LC.TTP	95
<i>New LC.TTP driver options</i>	95
<i>New compiler options</i>	99
Linker	101
CLINKWITH	102
ASM	105
Basic concepts	105
<i>Source format</i>	105
<i>Label</i>	106
<i>Operation</i>	106
<i>Operands</i>	106
<i>Number representations</i>	107
<i>Comment</i>	108
Addressing modes	108
The assembler from the command line	110
Assembler directives	113
<i>Conditional assembly</i>	118
Macro definition	120
Pre-defined macros and synonyms	121
Interfacing C with assembly language	122
<i>Control sections</i>	123
<i>Code sections</i>	123
<i>Data sections</i>	125
<i>BSS and offset sections</i>	127

<i>Function Entry Rules</i>	128
<i>Standard arguments</i>	129
<i>Register arguments</i>	130
<i>The <code>__asm</code> keyword</i>	131
<i>Function exit rules</i>	132
<i>Calling assembly from C</i>	133
<i>Calling C from assembly</i>	135
Asm error messages	137
DERCS	143
<i>Introduction</i>	143
<i>Running DERCS</i>	143
<i>Programming with DERCS</i>	145
CPXBUILD	147
<i>Introduction</i>	147
<i>Running CPXBUILD</i>	147
Appendix A	149
Notes	151

Lattice C 5.5

The Editor

The editor supplied with Lattice C is fully integrated with the system which means that you can develop programs in an intuitive and interactive manner, creating and editing your programs in the same environment as running and debugging your finished masterpiece.

Moreover, those of you with strong preferences for your own editor can dispense with the HiSoft editor and use your own favourite package along with the TTP version of Lattice C; although you will lose the benefits of interactive development.

The editor for Lattice C is a multi-window screen editor which allows you to enter and edit text and save and load from disk, as you would expect. It also lets you print some or all of your text, search and replace text patterns and use any of your computer's desk-accessories. It is GEM-based, which means it uses all the user-friendly features of GEM programs that you have become familiar with such as windows, menus and mice. However, if you're a die-hard used to the hostile world of computers before the advent of WIMPs, you'll be pleased to know you can do practically everything you'll want to do from the keyboard without having to touch a mouse.

The editor is 'RAM-based', which means that the file you are editing stays in memory for the whole time, so you don't have to wait while your disk grinds away loading different sections of the file as you edit. As the ST/TT range of computers have so much memory, the size limitations often found in older computer editors do not exist with Lattice C. As all editing operations, including things like searching, are RAM-based they act extremely quickly.

When you have typed in your programs it is not much use if you are unable to save them to disk, so the editor has a comprehensive range of save and load options, allowing you to save all or part of the text and to load other files into the middle of the current one, for example.

To get things to happen in the editor, there are various methods available to you. Features may be accessed in one or more of the following ways:

- Using a single key, such as a Function or cursor key;
- Clicking on a menu item, such as Save;
- Using a menu shortcut, by pressing the Alternate key (subsequently referred to as *Alt*) in conjunction with another, such as *Alt-F* for *Find*;
- Using the Control key (subsequently referred to as *Ctrl*) in conjunction with another, such as *Ctrl-A* for *cursor word left*;
- Clicking on the screen, such as in a scroll bar.

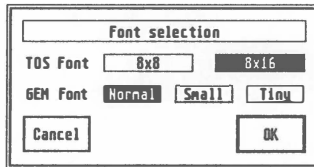
The menu shortcuts have been chosen to be, hopefully, easy to remember.

A word about pop-up menus and dialogs

The editor makes extensive use of dialog boxes and pop-up menus, so it is worth recalling how to use them, particularly for entering text. The editor's dialog boxes contain buttons, radio buttons, and editable text.

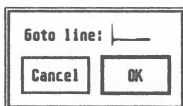
Exit buttons may be clicked on with the mouse and cause the dialog box to go away. Usually there is a default button, shown by having a wider border than the others. Pressing *Return* on the keyboard is equivalent to clicking on the default button. Where there are non-default buttons, the editor allows these to be selected from the keyboard using the sequence *Alt*-first letter of the button name; obviously where several buttons have the same first letter only one may be selected!

Radio buttons are groups of buttons of which only one may be selected at a time - clicking on one automatically de-selects all the others.



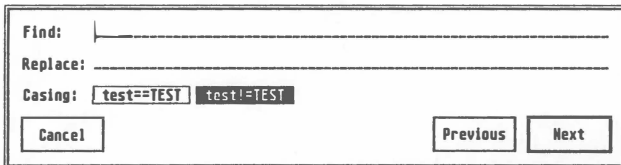
A dialog with buttons (OK, Cancel) and radio buttons (Normal, Small etc.)

Editable text is shown with a dotted line, and a vertical bar marks the cursor position.



Editable text

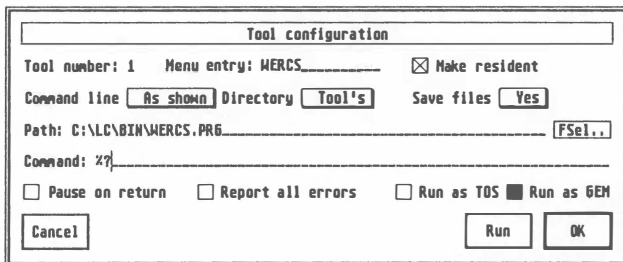
Characters may be typed in and corrected using the Backspace, Delete and cursor keys. You can clear the whole edit field by pressing the Esc key. If there is more than one editable text field in a dialog box, you can move between them using the Tab key or the ↓ and ↑ keys or by clicking near them with the mouse.



More than one editable text field

Some dialog boxes allow only a limited range of characters to be typed into them - for example the Goto... dialog box only allows numeric characters (digits) to be entered.

As well as the conventional GEM user interface facilities, the editor also uses some extensions. To illustrate these, consider the dialog box shown below:



The Tool Configuration dialog box

Some options are accessed via 'pop-up' menus similar to those used by Atari's new control panel. Thus if you move the mouse over the As shown selection (by Command line) and press down on the left mouse button, a menu like this will pop up:



A pop-up menu

This indicates that the current setting for this option is As shown. The mouse will highlight the current selection that you are making and when you let go of the mouse this indicates that you have made your selection. If you let go outside the pop-up menu then this is taken as cancelling the selection.

The box beside Make resident has a cross in it, indicating that this option is selected; similarly Report all errors is *not* selected. Clicking in one of these boxes, or the associated text, will cause that option to be toggled on and off.

Run as TOS and Run as GEM are a pair of 'radio options'; the solid box indicates the currently selected item: clicking on Run as TOS will change both boxes.

Some of the menu items on the main 'drop-down' menus now have sub-menus; these are indicated by a \diamond symbol. For example:

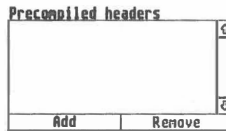


A sub menu

When you highlight a menu item (like Arrange Windows in the example above), the corresponding sub-menu will appear after a short delay. You can then move the mouse to the right to select the particular item that you want. To cancel the operation just let go of the mouse without selecting a sub-item or move to another item from the main menu.

If the editor doesn't have enough room to display the sub-menu to the right of the main menu, it will do so on the left; the items are selected in the same way.

The editor also uses a number of list boxes; these allow a number of selections to be entered (e.g. multiple `#include` directories, `#define` symbols etc.).



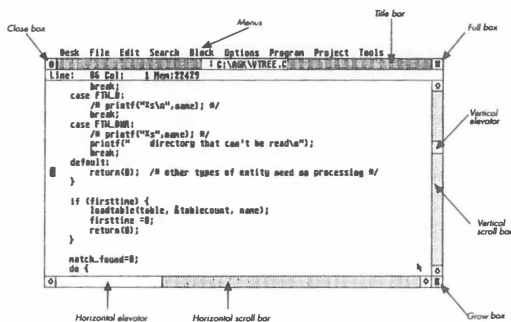
A list box

To add a new element to the list, click on the Add button, whilst an existing element may be removed by clicking on the item (which will become highlighted) and then clicking Remove. To edit an existing item, double-click on it.

The Editor's windows

Having loaded Lattice C, you will be presented with an empty window with a status line at the top and a flashing black block, which is the *cursor*, in the top left-hand corner.

The window used by the editor works like all other GEM windows, so you can move it around by using the *Title bar* on the top of it, you can change its size by dragging on the *Grow box*, and make it full size (and back again) by clicking on the *Full box*.



A GEM window

The status line contains information about the cursor position in the form of Line and Column offsets as well as the number of bytes of memory which are free to store your text. Initially this is displayed as 29980, as the default text size is 30000 bytes. You may change this default if you wish, together with various other options, by selecting Preferences, described later. The 'missing' 20 bytes are used by the editor for internal information. The rest of the status line area is used for error messages, which will usually be accompanied by a 'ping' noise to alert you. Any message that is printed will be removed when subsequently you press a key.

Switching Windows

The editor has support for up to seven windows, which can be selected by pressing Alt-1 to Alt-7 (on the top row of numbers, *not* on the numeric pad). The windows can be organised in a number of ways and you can select this using Arrange Windows on the Options menu. Try this out for yourself to get the idea of how the different arrangements work.

If you have a preferred window arrangement, you can get the editor to remember your preference by holding down Ctrl whilst selecting the layout. The layout will then become permanent and the editor will rearrange the windows as necessary to conform to your preference.

You can cycle through the open windows using the Cycle Windows command from the Edit menu (or use Ctrl-V), by clicking on the appropriate window with the mouse or by selecting the appropriate sub-item from the Window item on the Edit menu.

To close a window and thus free the memory used by it, click on its close box or use the Ctrl-W key combination.

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 the Cut command, switch windows (as described above) and then Paste. See below for more detail on cut and paste.

Entering text and moving the cursor

To enter text, simply type on the keyboard and at the end of each line press the Return key (or the Enter key on the numeric pad) to start the next line. You can correct your mistakes by pressing the Backspace key, which deletes the character to the left of the cursor, or the Delete key, which removes the character on the cursor.

Cursor keys

To move the cursor around the text to correct errors or enter new characters, you can use the cursor keys, labelled ← → ↑ and ↓ or the mouse; move the cursor to a specific position on the screen with the mouse pointer and click. If you position the cursor past the right-hand end of the line and type some text at that point the editor will automatically add the text to the real end of the line. If you type in long lines the window display will scroll sideways if required.

When you cursor up at the top of a window the display will either scroll down if there is a previous line, or print the message `Top of file` in the status line. Similarly if you cursor down off the bottom of the window the display will either scroll up if there is a following line, or print the message `End of file`.

You can move the cursor on a character basis by clicking on the arrow boxes at the end of the horizontal and vertical scroll bars.

To move immediately to the start of the current line, press `Ctrl ←`, and to move to the end of the current line press `Ctrl →`.

To move the cursor a word to the left, press `Shift ←` and to move a word to the right press `Shift →`. You cannot move past the end of a line with `Shift →`. A word is defined as anything surrounded by a space, a tab or a start or end of line. The keys `Ctrl-A` and `Ctrl-F` also move the cursor left and right on a word basis.

To move the cursor a page up, you can click on the upper grey part of the vertical scroll bar, or press `Shift ↑`. To move the cursor a page down, you can click on the lower grey part of the scroll bar, or press `Shift ↓`.

Tab key

The **Tab** key inserts a special character (ASCII code 9) into the buffer, which on the screen looks like a number of spaces, but is rather different. Pressing **Tab** aligns the cursor onto the next 'multiple of 4' column, so if you press it at the start of a line (column 1) the cursor moves to the next multiple of 4, +1, which is column 5. Tabs are very useful indeed for making items line up vertically and its main use in Lattice C is for such things as indenting structured program lines. When you delete a tab the line closes up as if a number of spaces had been removed. The advantage of tabs is that they take up only 1 byte of memory, but can show on screen as many more.

You can change the tab size before or after loading Lattice C; to change the default use the **Preferences** command described shortly.

Backspace key

The **Backspace** key removes the character to the left of the cursor. If you backspace at the very beginning of a line it will remove the 'invisible' carriage return and join the line to the end of the previous line. Backspacing when the cursor is past the end of the line will delete the last character on the line, unless the line is empty in which case it will re-position the cursor on the left of the screen.

Delete key

The **Delete** key removes the character under the cursor and has no effect if the cursor is past the end of the current line.

The Edit menu

Edit	
Cut	⌘F5
Copy	⌘F4
Paste	F5

ASCII Table... ⌘Ins	

Goto Top	⌘T
Goto Bottom	⌘B

Goto...	⌘G

Arrange Windows	⌘
Cycle Windows	⌘V
Window	⌘

The commands on the top of the Edit menu may be used to perform the conventional Cut, Copy and Paste operations on marked blocks.

These are described under *Block commands*, below.

Go to top of file

To move to the top of the text, click on Goto Top from the Edit menu, or press Alt-T. The screen will be re-drawn if necessary starting from line 1.

Go to end of file

To move the cursor to the start of the very last line of the text, click on Goto Bottom, or press Alt-B.

Goto line

To move the cursor to a specific line in the text, click on Goto... from the Edit menu, or press Alt-G. A dialog box will appear, allowing you to enter the required line number. Press Return or click on the OK button to go to the line or click on Cancel to abort the operation. After clicking on OK the cursor will move to the specified line, re-displaying if necessary, or give the error End of file if the line doesn't exist.

Another fast way of moving around the file is by dragging the slider on the vertical scroll bar, which works in the usual GEM fashion.

Block Commands

Block	
Block Start	F1
Block End	F2

Save Block	F3
Copy Block	F4

Delete Block	⌘F5

Remember Block	⌘F4
Paste Block	F5

Print Block	⌘M

A *block* is a marked section of text which may be copied to another section, deleted, printed or saved onto disk. Blocks may be marked using the mouse, via menu items or with function keys.

A marked block is highlighted by showing the text in reverse. While you are editing a line that is within a block this highlighting will not be shown but will be re-displayed when you leave that line or choose a command.

Marking a block

The simplest way to mark a block is to click on the first character in the block and drag the mouse to the end of the block. The block will be highlighted by showing the text in reverse as you drag the mouse. When you move the mouse to the bottom of the window, the window will scroll. Conversely, moving the mouse to the top of the window, will cause the window to scroll in the opposite direction. You may start marking a block, by clicking at the end if you wish.

Double-clicking will cause the word 'under' the mouse to be marked as the block. If you double-click and then drag, text will be highlighted a word at a time. Clicking in the the left hand margin of the window causes dragging to occur a line at a time.

The start of a block may also be marked by moving the cursor to the required place and selecting **Block Start** or pressing key F1. The end of a block can be marked by moving the cursor and selecting **Block End** or pressing key F2. The start and end of a block do not have to be marked in a specific order - if it is more convenient you may mark the end of the block first.

The Clipboard: Copy, Cut & Paste

Lattice C provides conventional clipboard facilities, as popularised by the Apple Macintosh. Once you have marked a block you may copy it to the clipboard by selecting **Copy** from the **Edit** menu. The main text will remain as it is. The contents of the clipboard may then be inserted at another position by moving the cursor there and selecting **Paste**.

The current block may be deleted using **Cut** from the **Edit** menu; selecting **Paste** will then insert the block that was cut (unless you have used **Copy** in the mean time). Thus to move a block with this method, **Cut** the block from its original position and then **Paste** it into its new one.

The block menu also gives you the flexibility of the following commands.

Saving a block

Once a block has been marked, it can be saved by clicking on **Save Block** from the **Block** menu or by pressing key **F3**. If no block is marked, the message **What blocks!** will appear. If the start of the block is textually after its end the message **Invalid block!** will appear. Both errors abort the command. Assuming a valid block has been marked, the **GEM** file selector will appear, allowing you to select a suitable disk and filename. If you save the block with a name that already exists the old version will be overwritten - no backups are made with this command.

Copying a block

A marked block may be copied, memory permitting, to another part of the text by moving the cursor to where you want the block copied and clicking on **Copy Block** or by pressing key **F4**. If you try to copy a block into a part of itself, the message **Invalid block!** will appear and the copy will be aborted.

Deleting a block

A marked block may be deleted from the text by clicking on **Delete Block** or by pressing **Shift-F5**. The shift key is deliberately required to prevent it being used accidentally. A deleted block is remembered, memory permitting, in the clipboard, for later use. This is equivalent to **Cut** on the **Edit** menu.

Copy block to block buffer

The current marked block may be copied to the block buffer, memory permitting, using **Remember Block** or by pressing **Shift-F4**. This can be very useful for moving blocks of text between different files by loading the first, marking a block, copying it to the block buffer then switching to another window or loading the other file and pasting the block buffer into it. This is equivalent to **Copy** on the **Edit** menu.

Pasting a block

A block in the clipboard may be pasted at the current cursor position by clicking on **Paste Block** or by pressing **F5**. This is equivalent to **Paste** on the **Edit** menu.



The contents of the clipboard is lost if the edit buffer size is changed and after a compilation.

Printing a block

A marked block may be sent to the printer by clicking on **Print Block** or by pressing **Alt-W**. An alert box will appear confirming the operation and clicking on **OK** will print the block. The printer port used will depend on the port chosen with the **Control Panel**, or will default to the parallel port. Tab characters are sent to the printer as a suitable number of spaces, so the net result will normally look better than if you print the file from the **Desktop**.



If you try to print when no block is marked at all then the whole file will be printed.

Block markers remain during all editing commands, moving where necessary, and are only reset by the commands Delete block and Load.

Delete text

Delete line

The current line can be deleted from the text by pressing Ctrl-Y.

Delete to end of line

The text from the cursor position to the end of the current line can be deleted by pressing Ctrl-Q

UnDelete Line

When a line is deleted using either of the above commands it is preserved in an internal buffer, and can be re-inserted into the text by pressing Ctrl-U, or the Undo key. This can be done as many times as required, particularly useful for repeating similar lines or swapping individual lines over.

Delete block

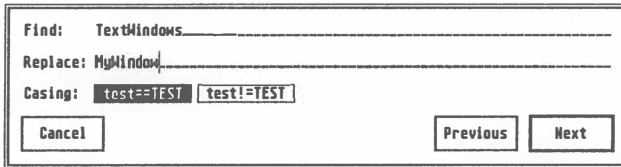
A marked block may be deleted from the text by clicking on Delete Block or by pressing Shift-F5. The shift key is deliberately required to prevent it being used accidentally. A deleted block is remembered, memory permitting, in the clipboard, for later use. This is equivalent to Cut on the Edit menu.

Searching and Replacing Text



The commands on the Search menu may be used for finding and perhaps replacing existing text. The strings involved are set up by selecting Find or press Alt-F.

This allows you to enter the find and replace strings as shown in the following dialog box:



In the example above TextWindows has been entered as the find string and MyWindow as the replace string.

If you click on Cancel, no action will be taken; if you click Next (or press Return) the search will start forwards, while clicking on Previous will start the search backwards. If you do not wish to replace, leave the replace string empty.

If the search is successful, the screen will be re-drawn with the cursor positioned at the start of the string. If the string could not be found, the message Not found will appear in the status area and the cursor will remain unmoved.

Whether test is treated as the same as TEST or Test etc. depends on which Casing button is selected. In the example above the search would stop if TEXTWINDOWS was found; if test!=Test was selected then the search would not find TEXTWINDOWS .

To find the next occurrence of the string click on Find Next from the Edit menu, or press Alt-N. The search starts at the position just past the cursor.

To search for the previous occurrence of the string click on Find Previous from the Search menu, or press Alt-P. The search starts at the position just before the cursor.

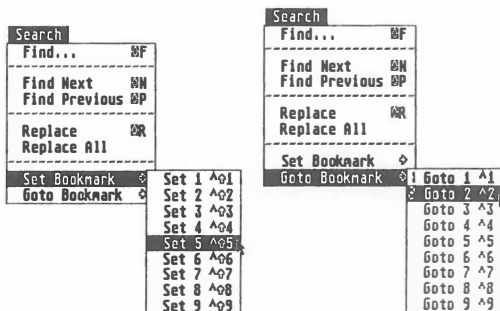
Having found an occurrence of the required text, it can be replaced with the replace string by clicking on Replace from the Search menu, or by pressing Alt-R. Having replaced it, the editor will then search for the next occurrence.

If you wish to replace every occurrence of the find string with the replace string from the cursor position onwards, click on Replace All from the Search menu. During the global replace the Esc key can be used to abort when the status area will show how many replacements were made. There is deliberately no keyboard equivalent for Replace All to prevent it being chosen accidentally.

To search and replace Tab characters press Ctrl-I when typing in the dialog box. Other control characters may be searched for in a similar manner except for the CR (Ctrl-M) and LF (Ctrl-J) characters. Alternatively, press Shift-Ins and this will display the character set from which you may pick the required character with the mouse.

Bookmarks

A further way to navigate your source text is via the use of bookmarks. A bookmark is set by selecting the appropriate Set Bookmark item from the Search menu or by using Ctrl-Shift and a digit key (not the numeric keypad). When you set a bookmark the corresponding item on the Goto Bookmark menu will become enabled. Then, selecting this item, or by pressing Ctrl and the digit, will return you to the original position.



When you set a bookmark, the window number to which it refers is displayed in the menu. Going to a bookmark may cause you to switch windows. Note that bookmarks that are set in a given window are lost when you close that window.

Disk Operations

File	
New	
Load...	⌘L
Insert File	⌘I
Revert	
Close	⌘W
Save	⌘S
Save As...	⌘S

Delete File	
Change Directory	

Quit	⌘Q

The File menu contains many operations that involve using the disk system; you can save and load your source file, insert text into your source, delete a file from a disk and more.

New

Select New to open an empty window, assuming that there is one available - you are allowed up to seven windows at once in Lattice C.

Assuming that there are no more than six windows open, New will create a window which is empty and has no title.

Loading Text

To load in a new text file, click on Load from the File menu, or press Alt-L. This will open a new window (and warn you if no more windows are available) or select an unused window and then a file selector will appear, allowing you to specify the disk and filename. Assuming you do not Cancel, the editor will attempt to load the file. If it will fit, the file is loaded into memory and the new window is re-drawn. If it will not fit an alert box will appear warning you, and you should use Preferences to make the edit buffer size larger, then try to load it again.

If the file can't be found a dialog box will appear, asking you if you wish to create that file. You may do so, or alternatively modify the filename and try again.

When loading Lattice C from a CLI, you may include up to seven filenames. The corresponding files will then be loaded automatically. If a file cannot be found you will be asked if you wish to create it or may change the filename if you wish. If you use the desktop to install Lattice C as a *GEM takes parameters* (GTP) program then you may also enter up to seven file names to be loaded.

Revert

Revert will warn you that you are about to lose the text in the selected window and, assuming that you choose to continue, it will then re-load the last saved version of the file that you were editing in this window.

Revert will do nothing if you try to use it on a file that has not been saved previously.

Save As...

To save the text you are editing, click on **Save As...** from the **File** menu, or press **Alt-S**. The **File Selector** will appear, allowing you to select a suitable disk and filename. Clicking **OK** or pressing **Return** will then save the file onto the disk.

If you click on **Cancel** the text will not be saved. Normally if a file exists with the same name it will be deleted and replaced with the new version, but if **Make backups** is selected from **Preferences** then any existing file will be renamed with the extension **.BAK** (deleting any existing **.BAK** file) before the new version is saved.

Save

If you have already done a **Save As** (or a **Load**), the editor will remember the name of the file and display it in the title bar of the window. If you want to save it without having to bother with the file selector, you can click on **Save** on the **File** menu, or press **Shift-Alt-S**, and it will use the old name and save it as above. If you try to **Save** without having previously specified a filename you will be presented with the **File Selector**, as in **Save As**.

Inserting Text

To read a file from disk and insert it at the current position in your text, click on **Insert File** from the **File** menu, or press **Alt-I**. The **File Selector** will appear and assuming that you do not cancel, the file will be read from the disk and inserted, memory permitting.

Delete File

You may want to delete a file from disk (if for instance you have run out of disk space whilst trying to save); click on Delete File. The File Selector will appear, allowing you to select a suitable disk and filename. Clicking OK or pressing Return will then delete the file from the disk. If you click on Cancel the file will *not* be deleted.

Close

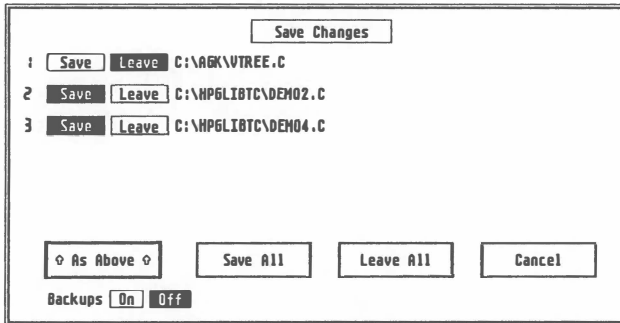
This is the same as pressing Ctrl-W and will close the currently selected window. If the file that is being edited in this window has been changed since it was loaded or is a new file, you will be warned before the window is closed. You can choose to continue and lose your changes, cancel the action or save the changes.

Change Directory

This option allows you to move the current directory path; this can be useful when running programs which expect all of their files to be in the same place as the program itself. After clicking on Change Directory the File Selector will appear, allowing you to select a suitable disk and folder name. Clicking OK or pressing Return will then change the directory. If you click on Cancel the directory path will not be changed.

Quitting Lattice C

To leave Lattice C, click on **Quit** from the **File** menu, or press **Alt-Q**. If changes have been made to the text which have not been saved to disk, an alert box will appear asking for confirmation.



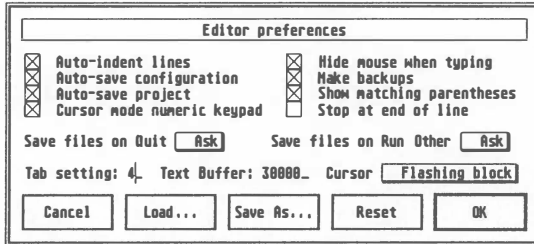
This example shows that two files have changed. Clicking on **Save All**, **As Above** or pressing **Return** will exit the editor saving the changes. Clicking on **Cancel** will return to the editor. **Leave All** will ignore all the changes you have made.

If you wish to save some files but not others click on the appropriate **Leave** buttons. For example if you clicked on the **Leave** button by **VTREE.C** in the above example and then pressed **Return**, only **DEMO2.C** and **DEMO4.C** would be saved.

You can also enable and disable backups from this dialog box. This is useful if you normally use backups, but decide that you don't require a backup of a one line change.

Configuring the editor

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



The editor preferences box

This box allows you to set up the editor as you would like to use it; you can then save your customisation to disk so that the editor will always behave the same way. Here are the different settings that you can change.

Auto-indent lines

Selecting this option sets 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. This allows you to lay out your program neatly, by simply pressing Return.

Auto-save configuration

When this option is selected, the current preferences will automatically be saved when you exit the editor. So when you load the editor again, the preferences will be just the same as when you last used it.

Auto-save project

When this option is selected, the current project will automatically be saved when you exit the editor. So when you load the editor again, the project (including the compiler's options) will be just the same as when you last used it.

Cursor Mode Numeric pad

The Cursor Mode 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 diagram below:



When this option is not selected the keyboard reverts to returning the digits etc.

Hide mouse when typing

Selecting Hide mouse when typing causes the mouse pointer to disappear when you start entering text with the keyboard. As soon as you move the mouse, or use a command that displays a dialog box, the mouse will re-appear. This option may be disabled if you prefer to always see the mouse on the screen.

Make Backups

Selecting this option causes the editor to make a backup (with the extension .BAK) when saving files.

Show matching parentheses

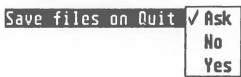
This facility lets you check that your parentheses match. With this option enabled, when you press) the cursor will quickly move to any matching (character and then back to the current position, thus you can ensure that you have closed the correct number of brackets in a complex expression. If you find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

Stop at End of Line

When this option is selected, if you press cursor left at the beginning of a line or cursor right at the end of line, the cursor does not move. Disabling this option, 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.

Save files on Quit



By default the editor will prompt you, if you are about to quit without having saved all the files, you have changed.

The saving of these files can be made automatic by selecting Yes or disabled by selecting NO (but don't blame us if you forget to save your files!).

Save files on run other

This enables you to choose whether files are saved before using the Run Other and Run with Shell commands, in the same way as that for Save files on Quit.

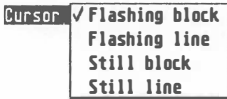
Tab setting

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 10000 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. Care should be taken to leave sufficient room in memory for compilations - pressing the Help key displays free system memory, and for compilations this should always be at least 100k bytes. 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 been saved.

Cursor



By default the editor cursor is a flashing block, but this can be changed as required.

Load...

This button lets you load a settings file. The editor settings are normally stored in a file called HISOFTED.INF in the current directory, but the editor will 'look down' both the AES and GEMDOS paths. If you want to use more than one set of preferences, then you can explicitly load a settings file.

Saving preferences

To save the settings file you can either choose Save as... from the Preferences box or choose Save preferences from the Options menu.

This latter command, on the Options menu, saves the current editor, compilation and Tools menu preferences under the name HISOFTED.INF. If you want to call your settings file a different name you should use Save as... in the Preferences... box, as described below.

When the editor is loaded, it looks for the HISOFTED.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 this way will put the .INF file in the same place it was loaded from or, if it was not found, it will be placed in the current directory path.

In addition to saving the editor configuration the current program buffer size, from within the compilation options dialog box, is also saved.

Use *Save as...* from the Preferences box to save a settings file with a name other than HISOFTED.INF; an extension of .INF is still usual.

With this option you can save a number of different settings files under different names; however the editor always loads the settings file called HISOFTED.INF when it starts up so that, if you want to make a particular settings file the default, you will need to re-name it to HISOFTED.INF.

Reset

Clicking on this box causes the settings to be reset to their default values; useful if you have made a complete mess of your options.

Running other programs

There are three ways that you can execute other programs from within the editor; *Run Other...*, *Run with Shell...* and by a selection from the *Tools* menu. These different methods will now be described.

Tools Menu

Tools	
MERCS	01
CPX-Build	02
Tool 3	03
Tool 4	04
Tool 5	05
Tool 6	06
Tool 7	07
Tool 8	08
Tool 9	09
Tool 10	0A
Tool 11	0B
Tool 12	0C
Tool 13	0D
Tool 14	0E
Tool 15	0F
Tool 16	10
Tool 17	11
Tool 18	12
Tool 19	13
Tool 20	14

Run Other...	00
Run with Shell	00

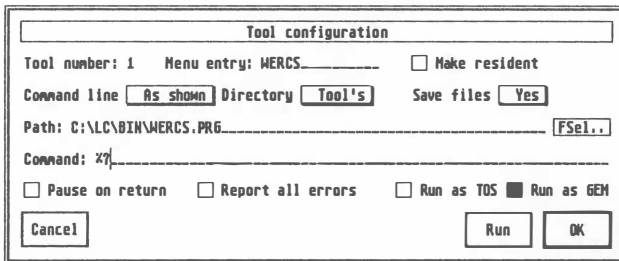
The Tools menu lets you run programs of your choice from within the editor using a single keystroke or click of the mouse.

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

The preferences file that we supply is already set up to run 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:



The dialog box is titled "Tool configuration" and contains the following fields and options:

- Tool number: 1
- Menu entry: MERCS
- Make resident
- Command line: Directory: Save files: Yes
- Path: C:\LCC\BIN\MERCS.PRG
- Command: Z?
- Pause on return
- Report all errors
- Run as TOS
- Run as GEM
-

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 FSe1 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.

Command 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 As shown 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, as in the example above.

Directory



This sets up which directory will be the current one when the tool is run. Current will leave the directory as that of the editor itself.

Tool's switches to the directory of the tool being run, whereas Top window switches to where the file in the current window is stored on disk.

Save Files

This option changes 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 Lattice C.

Path

This option specifies which program is actually to be run. If you give a full pathname, or select one by clicking on the FSel.. button then that specific file is run. If you just use a name then this will be treated as if you had used it as an argument to the Run with Shell command described above.

Pause on return

This option controls whether the editor pauses after running the tool. Typically you will select this when running a TOS program but disable it when running a GEM program.

Report all errors

This option allows you to specify which errors the editor will bring to your attention when returning. If this option is not selected then you will only be alerted to negative return codes from programs, i.e. those normally indicating GEMDOS errors. Selecting it will also force positive program error returns to be flagged.

Run as TOS & Run as GEM

These buttons select 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.

Make Resident

If this item is selected then when the editor next loads it will attempt to load this tool into memory and make it resident, i.e. merely execute the tool from memory rather than load it from disk each time. This is particularly useful with substantial programs like WERCS.

As well as the obvious disadvantage of permanently tying up your memory, not all programs can be made resident. In general your own Lattice C programs can be made resident if compiled using the Resident startup option.

We do not recommend running third party programs in this way. They may crash immediately, or the second time they are run or may simply not quite work correctly possibly destroying your valuable files in the process.

Running Tools

Running 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.

Run Other...

This command, on the Tools menu (also reached by Alt-0), lets you run other programs from within the editor, then return to it when they finish.

When you select Run Other... you will first be warned if you have not saved your source code (unless you have modified the setting of the Save files on Run Other option in Preferences). 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' execution 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 (remember, you are prompted for a command line) these characters will be replaced by the full name of the file that you are currently editing. To pass the name without its extension, use `%?`.

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

Run with Shell...

This command also 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-0.

It differs from Run Other in that 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, PKS and Gulam shells 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, which can be set using the Environment command from the Options menu.

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.

Setting the Path

The editor maintains a number of directory paths to make the operation of the integrated environment natural and seamless.

Paths are routes to files. Normally you keep all files of a similar type or usage in one folder or you may have a number of related folders all within one outer folder. For example you probably have an LC folder containing the Lattice C program, its tools and its libraries.

In order that a program that uses these files can find them without having to ask the user for help, both the ST/TT operating system and the Lattice C editor maintain a number of directory paths, some of which you can alter.

Here is a summary of the paths used by the integrated environment, how they are set and what uses them:

Current directory - this is a path that is set up (initially) by the program which ran the current program. For example, for the Lattice C editor this path will have been set up by the Desktop, assuming of course you ran Lattice C from the Desktop. However, since the editor allows this to be changed (via the Change Directory command on the File menu), it is normally reset to whatever was last stored in the HISOFTED.INF file, to save you having to change it every time you run the editor.

Most of the disk-related functions within the editor will search this path first.

GEMDOS path - this path is that contained in the PATH environment variable. It is used by shells (e.g. Craft, PKS Shell, Gulam) to locate programs to run. It is specified as a list of , or ; separated folder names, each of which specify a folder which should be searched when trying to locate a file.

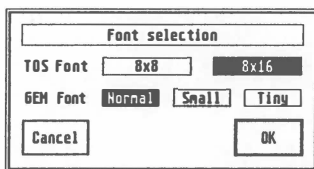
Within the editor it is used by Run with Shell, to locate the named program, and initially when attempting to find the LC1.LC file. Other tools, like WERCS, may use it for locating subsidiary files, such as WERCS.RSC and WERCS.INF.

AES path - this is the path used by the AES when the user calls one of the AES routines which search for a file (*shel_find* and *rsrc_load*). Internally the format of this variable is identical to the *GEMDOS* path (in fact it *is* the *GEMDOS* PATH for the AES program!), although the AES provides no way of altering it and merely sets it to *A:* for a floppy based machine or *C:* for a hard disk machine.

Miscellaneous Commands

Fonts...

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



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

In ST 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. In TT medium resolution, using 8x8 will give 60 lines instead of 30.

You should be aware that any change of font that you make here will also be effective outside the editor, after you leave it.

ASCII Table...

To be found on the Edit menu, this displays a pop-up dialog box at the current mouse position, showing all the ASCII characters:

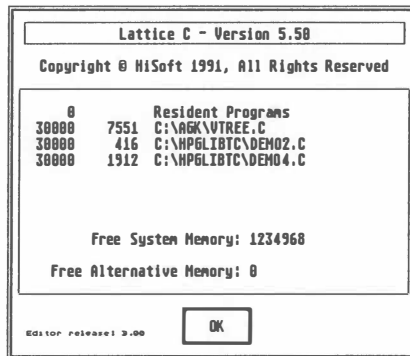


You may click on an individual character and it will be added to the text that you are editing at the current cursor position. You can bring up this display from the keyboard using Shift-Insert. This short cut can also be used in the editor's dialog boxes.

Note that the characters that would confuse the editor are 'greyed out' and may not be selected. Remember that characters other than those in the standard 7 bit ASCII set are not necessarily the same on other computers.

About Lattice C

If you select About Lattice C... from the Desk menu, a dialog box will appear giving various details about Lattice C, including its version number. You will also be told the amount of free memory that is available to you and how much is used by the resident programs including the text in the open windows.



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 cursor and function keys, as well as the free memory left for the system.

Desk Accessories

If your 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 an editor window, which brings the editor window to the front; you can then re-size it so you can see some part of the control panel's window behind it. When you want to bring the control panel back 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 Launching

You may configure Lattice C to be loaded automatically whenever a source file is double-clicked from the Desktop, using the *Install Application* option.

To do this you first have to decide on the extension you are going to use for your files, which we recommend to be .C for C files. Having done this, go to the Desktop, and click once on LC5.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 (or whatever you require), and leave the GEM radio button selected. Finally click on the OK button (if you press Return it will be taken as Cancel).

Having done this, you will return to the Desktop. To test the installation, double-click on a file with the chosen extension which must be on the same disk and in the same folder as Lattice C and the Desktop will load Lattice C, which will in turn load in the file of your choice ready for editing or compilation.

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

Compiling Programs

Having produced your C program and saved it to disk using the editor you can then compile it. Normally this is a two stage process; first the compiler turns the C source file into an object file and then the linker links this object (together with any other object and/or libraries required) to produce an executable program.

Because most larger programs consist of more than one C source file, the integrated compiler provides a *Project manager* to aid in the maintenance of larger projects. For small, single file projects, the project manager is still employed using a *default* project.

The Project menu

Project	
Edit "DEFAULT"...	⌘E

New...	
Load...	

Save "DEFAULT"	
Save As...	

Link "DEFAULT"	
Make "DEFAULT"	⌘M
Make all "DEFAULT"	

Directory	◇
Run "DEFAULT"	⌘X
Debug "DEFAULT"	⌘D

The Project menu allows the entire structure of an application to be set up and managed from within the integrated compiler.

A project file (.PRJ) contains a list of all the files (and which files they include) and options which are required to rebuild an application.

Associated with each project is a project directory, the directory in which the project file is located. This directory is used as the current directory for both compiling and running the program, thus ensuring that subsidiary files (such as resource files) can be easily located.

New...

The **New...** option is used to create a new project, and presents a file selector to allow you select the project directory, and the name of the project file. Note that you should decide on a name for your project at this time to ensure that output file names are chosen appropriately.

Load...

The **Load...** option reloads a previously saved project. On loading the project the current in-memory project definition is completely replaced by the new one, together with all its options.

Save "..."

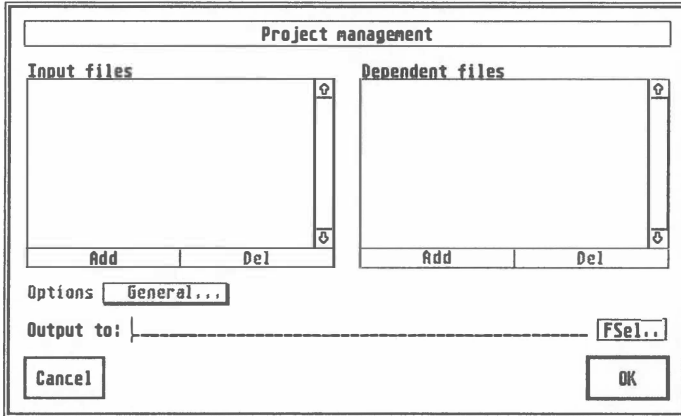
This saves the current project under the name originally given to the project.

Save As ...

Save As... is used to save the current project under a new name; this may be useful if you are editing an old project to create a new project.

Edit "..."

The Edit... item is the central part of the project interface. It allows the files which form part of a project to be set up, and the final output file to be set. The following large dialog box is used:



The main part of the Project management dialog consists of two list boxes. The left hand one is used to enter the source files which make up the project; the files may be added, deleted or changed in the normal list box manner. In addition it is possible to change the 'build order' of the files (i.e. the order in which they are built & then linked) by clicking on an entry and dragging it to a new position.

Within a project any number of input files may be specified in the Input files list; the extension of the file is used to decide how the project manager will process it:

Extension	Action
C	€
None	Compile named file
S	Assemble named file
LIB	Include named library file when linking
LNK	Include file as WITH file during linking
O	Include named object file when linking
Other	
PRJ	Make sub-project substituting target name in source file list

For each source file (.C/ .S) within a project dependent files may be specified; these are the files which are *included* by the main source file. To specify dependent files for a particular source file, selecting the source file will activate the Dependent files list box allowing any number of dependents to be entered.

In addition to allowing dependent files a source file may have specific options set. Note that this is only required where a particular source file should have specific options, normally the options for the whole project should be set via the normal options menu. To set a set of file specific options first select the file and then select the appropriate options box which you wish to change via the Optionspopup menu.

The output filename may be specified in the Output to line or via a file selector after clicking on FSel. The extension of the output file name is used by the project manager to decide what sort of object file is to be built:

Extension	Action
LIB	Pass input files to librarian
0	Pass input files to linker for pre-linking (PRELINK keyword)
Other	Pass input files to linker and any other automatic files (startup stubs, math libraries, gem libraries & C libraries) for linking into an executable.

Note that the project manager recognises any non .LIB or .0 extension as a normal executable file. It is often easiest to give the output file no extension and instead allow one to be invented based upon the setting of the *Executable options - Application type* setting.

Make "..."

Make "." is used to start the process of rebuilding a project. The project manager examines the time and dates of all the input source files with respect to their object files and rebuilds only those that have changed.

Note that you *must* ensure that the real time clock in your machine is set; if you do not the project manager will not function correctly.

Make all "..."

Make all "... is used to unconditionally rebuild a project. This can be useful if you have changed a global option which would affect the 'code model' and so all source files need rebuilding. This is obviously much slower than the Make "... option!

Link "..."

Link "... is used to unconditionally rebuild the output file; note that all the input files must be available, the project manager will not attempt to build any of them. Link "... is useful in situations where you have simply changed one of the linker options (e.g. *Linker options - Add exported symbols*) and want the project manager to build a new output file.

Note that the name Link "... is something of a misnomer; if the output file is a library (*output.lib*) then the librarian would be run instead!

Run "..."

Run "... runs the output file for the current project. Note that the output file must be executable in order to select this option; if the output file ends in .O or .LIB you will not be able to run it directly.

Debug "..."

Debug "... passes the output file for the current project to the debugger. The options to the debugger may be set using the *Options - Debugger...* option.

Note that the output file must be executable in order to select this option; if the output file ends in .O or .LIB you will not be able to run it directly.

Directory



Directory is used to select which directory is current when a program is run.

Because many GEM programs have difficulty finding subsidiary files when run from remote directories the default setting of *Directory - Project* ensures that the current directory is that of the project. For TOS or more robust GEM programs the *Directory - Current* is useful to ensure that a program will function correctly when run from a remote directory.

Problems

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.

Compilation Errors

When the compiler detects an error or something that may be an error (a warning) it generates a message; these errors are remembered, and can be recalled from the editor.

When you return to be editor you can use Alt-J to move to the next error with the error message displayed in the status line. If you have a large number of errors the editor may not be able to remember them all. Alt-J goes to the next error regardless of the position of the cursor; it will switch windows if required. To go to a previous error use Ctrl-J. The editor takes account of any insertions or deletions automatically so that unless one error (like a mistake in a definition) has caused multiple errors you should only need to compile once.

There's also the Shift-Alt-J command which finds the next error after the cursor in the current window. It is the appropriate one to use if you have got a number of include files and want to fix all the errors in one file before going on to the next one. You can also use it to find the first error in a file by typing Alt-T (to go to the top) and then Shift-Alt-J.

Occasionally the compiler will spot errors somewhat later than you might expect. This is usually because the text up to the point it has read is allowed in a certain context. If you have missed something out at the end of a line, then the error may be detected at the beginning of the next line.

On occasions the compiler will generate more than one error message as a result of a single error in your program; do not be put off by this. If you get confused, just re-compile.

Incidentally, if you start a compilation of a large program you can break out and returned to the editor using the key combination Left-Right-Shift when using the integrated compiler.

The Program menu

Program	
Assemble	⌘A
Check	⌘Y
Compile	⌘C
Pre-compile	
Pre-process	
Prototype...	

Previous error	⌘J
Next error	⌘J

The Program menu provides facilities for compiling single source files in a 'one-shot' manner. Although this menu allows compilation and assembly of source files, normally you will not use these functions, preferring instead the facilities of the Project menu.

Assemble

The Assemble option is used to assemble the current window, using the global project options (described below). Since an assembly language source file will normally be part of a much larger project it is normally easier to place the description of the file within a project for later reuse.

Check

The Check option, checks the syntax of the source text that is currently being edited without producing any output file. If the compiler is already loaded then this lets you check your program quickly.

Compile

The Compile option is used to compile the current window, using the global project options (described below). Since many C source files will be part of a much larger project it is normally easier to place the description of the file within a project for later reuse.

Pre-compile

The Pre-Compile option is used to generate a pre-compiled header file suitable for inclusion in the Compiler options - Advanced (Precompiled headers) dialog. The current source window is processed together with the global project options (described below) to produce a symbol file *source.sym*.

Note that any files mentioned in the Compiler options - Advanced (Precompiled headers) dialog *will* be loaded as part of the precompilation process and so must be available (i.e. if you are rebuilding an existing pre-compiled header you should remove its declaration from the Precompiled headers list).

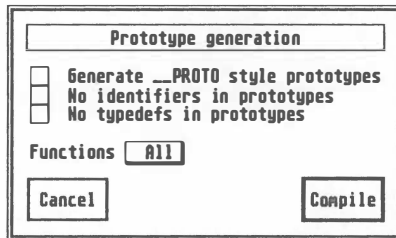
Pre-process

The Pre-process option is used to force the compiler to write the results of preprocessing the current source file into the output file *source.p*.

Prototype...

The Prototype... option is to invoke the compilers prototype generation option on the current window, using the global project options (described below), building a prototype file *source.i*.

This option produces a dialog allowing the options for the prototype generation to be set:



Generate `__PROTO` style prototypes **-pp**

This option forces the compiler to generate prototypes 'protected' by an `__PROTO` macro. Normally on re-reading the prototype file the prototypes will be used, however if the symbol `__NOPROTO` is pre-defined by the user then the prototypes will be ignored. This can be useful to allow portability of source files to older K&R compilers.

No identifiers in prototypes **-pi**

No identifiers in prototypes (-pi) suppresses the compilers generation of formal names for the prototypes. Since these names are not required in the prototypes removing them can save both space and compilation time later.

No typedefs in prototypes **-pt**

No typedefs in prototypes (-pt) suppresses the compilers use of typedef names within prototypes. In many cases this is desirable as the typedef may not be 'visible' to the prototype file and so any unknown types would cause the compiler to issue an error.

Functions



The *Functions* option allows the user to configure how prototyping deals with static functions.

There are three possible options:

All **-pr**

Causes the compiler to generate a prototype file containing prototypes for all functions defined in the source file.

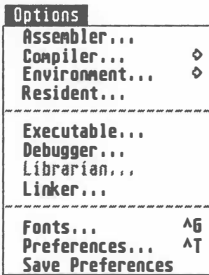
External **-pe**

Eliminates prototypes for all static functions. Only those functions available externally will have prototypes generated for them.

Static **-ps**

Generates prototypes only for static functions. Only those functions defined with the static function will be output.

The Options menu



The Options menu provides the main place from which options for both the compiler and editor are set.

The items on this menu relating to the compiler are described in this section; the other options are described elsewhere.

Environment...

The Environment... option allows the environment variables used by the tools which are run (and other parts of the Lattice C system as described above) to be altered.

The 'environment variables' are an array of strings which GEMDOS creates for every program which it runs. This array of strings contains 'variables' of the form *name=value* which the program may interrogate to obtain information about the 'environment' in which it is running (hence the name). Some typical environment variables are: PATH - specify the directories in which a program should look, INCLUDE - specify the directories in which a compiler should look for include files, EDITOR - the users preferred text editor which an application should run.

Normally a parent program creates a new environment for the child program; if this is not done explicitly then the child 'inherits' a copy of the parent's environment. Hence every program has its own, unique, copy of the environment.

For programs started normally from the Desktop, only a single environment variable is available, PATH.

Lattice C and the environment



Several environment variables are used by the compiler to locate files it may need.

These are INCLUDE, LIB, PATH and QUAD.

PATH

Executable path

This variable defines a sequence of directory prefixes where a program (e.g. the editor!) should search for an executable file (e.g. when trying to run LC1, LC2, GO etc.).

On selecting the Environment - 'PATH' option, the following dialog appears (although without all the entries!):



This shows a list of all the directory prefixes specified in the PATH variable. The entries in this list dialog may be manipulated in the manner described in the section *A word about pop-up menus and dialog*.

When a new entry is added, or an existing one edited (by double-clicking), a file selector is presented to allow the entry of a path.

INCLUDE

Include path

The INCLUDE variable is similar to the PATH variable except that it is used by the compiler to locate include files referenced by your program.

Manipulating this variable is done in an identical manner to that described for the PATH variable.

LIB

Library path

The LIB variable is similar to the PATH variable except that it is used by the linker to locate input and library files referenced by the editor.

Manipulating this variable is done in an identical manner to that described for the PATH variable.

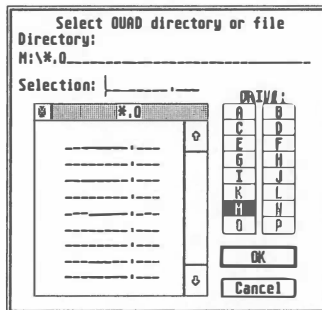
Note that just because a library file is in the library directory does not mean that the file will be linked in, you must tell the compiler to link it!

QUAD

Quad file

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.

On selecting the Environment - 'QUAD' option, a standard file selector appears to allow you to select a quad file name, or path. To select a specific name, enter it in the file selector, or to set a directory, leave the filename portion blank:

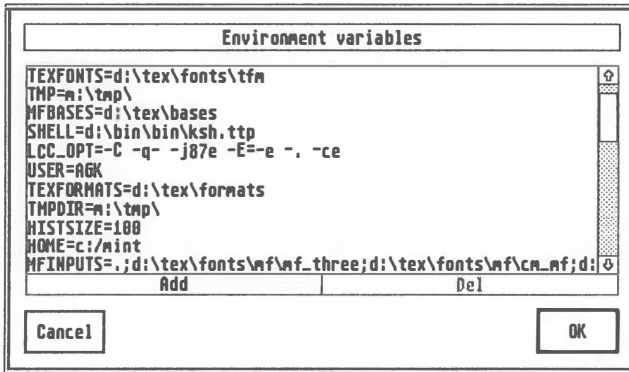


If you have a RAM disk installed you can greatly increase compiler performance if you use this as the quad temporary directory.

General...

Because the Lattice C editor provides a full visual shell, in order that other programs which may be run from it (e.g. the Tools, etc.), full support is available for arbitrary environment variables (i.e. other than PATH, INCLUDE, LIB and QUAD).

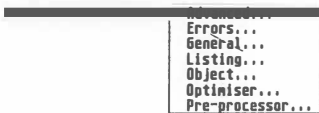
On selecting the Environment-General... option, the following dialog appears (although without all the entries!):



This shows a list of all environment variables known to the editor, either because they were inherited from the parent, or because you have previously set them. The entries in this list dialog may be manipulated in the manner described in the section *A word about pop-up menus and dialog*.

Note that the editor has no way of knowing which variables are list-type variables (like PATH) and which are single assignment type (like QUAD), so it is up to you to format the entries correctly for the tools you intend to run.

Compiler options

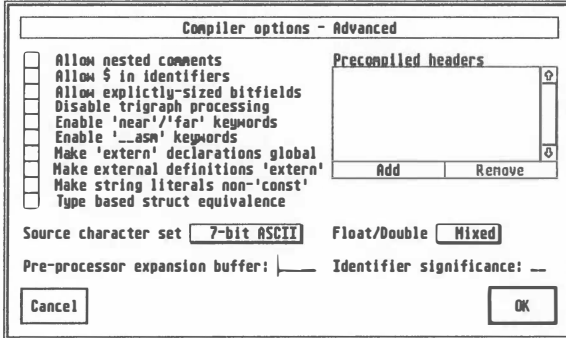


The Compiler options item and its associated sub-menu are the main place (for a project or single file) in which compilation options are set.

On selecting one of these options, one of a number of dialog boxes are used to allow the options to be set. Within the following descriptions the text of the option is shown on the left of the heading, whilst the command line option (for LC.TTP users) is shown on the right.

Compiler options - Advanced

The Advanced dialog includes all the options which are of use to experienced programmers who wish to tailor the translation environment to their needs.



Allow nested comments **-cc**

Allow \$ in identifiers **-cd**

Allow explicitly-sized bitfields **-cb**

This option allows the compiler to recognise bitfield specifications using types `char`, `short` and `long` rather than just the ANSI types of `int`, `unsigned int` and `signed int`.

Disable trigraph processing **-cg**

Disable trigraph recognition;. Trigraphs are enabled in *Strict ANSI mode* (`-ca`); we strongly recommend that you keep trigraphs disabled due to the overhead incurred by their recognition.

Enable 'near'/'far' keywords **-ck**

Enables the presence of the `near` and `far` keywords even when the *Strict ANSI mode* (`-ca`) has been specified.

Enable '__asm' keywords **-cr**

Enables the register keywords `__d0...__d7`, `__a0...__a7` and `__fp0...__fp7`, even when *Strict ANSI mode* (`-ca`) has been specified.

Make 'extern' declarations global **-cx**

Cause all `extern` declarations to have global scope. The compiler deals with implicit and explicit in-block (cf. global) `extern` declarations according to ANSI, i.e. their scope is restricted to that of the block. This option can be used to force their scope to global, for compatibility with older non-ANSI compilers.

Make external definitions 'extern' **-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. This 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.

Make string literals non-'const' **-ch**

When using the *Merge identical strings* (`-cs`) option the compiler gives each literal string the type `const char []`, although this can create many warnings which were not intended. This option disables this behaviour, and makes string literals of type `char []`.

Type based struct equivalence **-cq**

Relax the aggregate type checker to allow aggregates with common initial subsequences to type check equivalent. This means that `struct A` will be considered equivalent to `struct B` if the types of `struct A`'s members match the types of `struct B`'s members over the length of `struct B`. This option is most useful where a larger structure (`struct A`) has a smaller structure (`struct B`) embedded at the start of it and you wish to pass a pointer to `struct A` to a function expecting a pointer to a `struct B`.

Source character set **-e**



This option is used to activate the extended character set, either for 8-bit ability or to access codes used in Asian-language applications.

Use of any of these options *disables* compressed header files, and so the full uncompressed versions must be used.

Certain Asian characters are represented by two consecutive bytes, the first byte of which has its high bit set, i.e. a value above 127. With the *Source character set* (-e) option, a two-byte Asian character is treated as a unit in string and character constants. This means that when the compiler scans a text string enclosed in double quotes, it will recognise the first byte of an Asian character and suppress lexical analysis of the second byte. So, if the second byte is a backslash or a double quote, it will not receive any special processing.

Also, if you enclose an Asian character in single quotes, the compiler will produce a two-byte constant, and if you have not specified *Allow multi-character constants* (-cm), it will warn you that multi-character constants are not allowed.

8-bit Atari **-e**

8-bit coding is used. This means that characters above 0x80 are treated as normal characters, thus allowing all 8-bit ASCII codes to be embedded in a program.

Japanese **-e0**

Japanese coding is used. This means that characters with values from 0x81 to 0x9F and 0xE0 to 0xFC are treated as the start of a two-byte sequence. The characters from 0xA0 to 0xDF are single-byte Katakana codes.

Korean **-e2**

Korean character coding is used. This means that characters with values from 0x81 to 0xFD are treated as the start of a two-byte sequence.

Taiwanese **-e1**

Chinese/Taiwanese coding is used. This means that characters with values from 0x81 to 0xFC are treated as the start of a two-byte sequence.

Float/double **-f**



These options allow control over the precision attributed to float and double declarations used within the user code.

All Double **-fd**

Causes the compiler to treat all declarations as double precision.

All float **-fs**

Causes the compiler to treat all declarations as single precision.

Mixed **-fm**

Causes the compiler to treat `float` as single precision and `double` as double precision; .this is the default option.

Pre-processor expansion buffer: size **-zsize**

Whilst pre-processing the source file the compiler uses several buffers to store the pre-processed line. If this buffer overflows (giving a line buffer overflow or pre-processor symbol loop (macro expansion too long or circular) error) then the size may be increased using this option. The buffer has, by default, a size of 6000 bytes for 'big' compilers, or 3000 bytes for 'small' compilers

Precompiled headers **-Hfile.sym**

This list box specifies the precompiled header files the compiler is to pre-load into its symbol table. Precompiled header files are generated using the Pre-compile menu option.

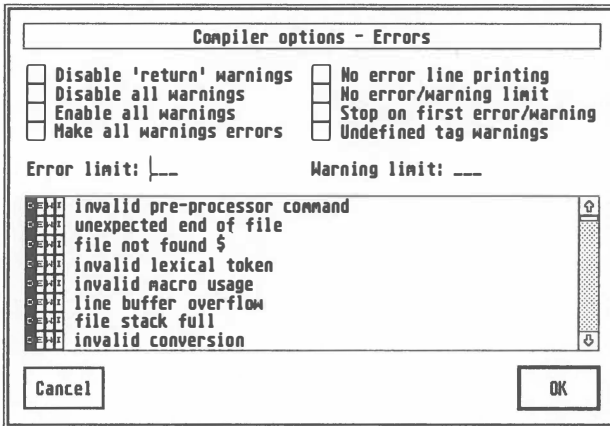
There is no limit to the number of precompiled header files that may be read in.

Identifier significance: sig **-nsig**

This option specifies the number of characters the compiler is to retain for identifiers. This can be useful if more than the default of 31 is required, or to reduce to 7 or 8 for compatibility with very old programs.

Compiler options - Errors

The Errors dialog provides options for controlling the errors reported by the compiler. This can be useful to enable additional diagnostic messages, or to suppress messages which are of no interest.



Disable 'return' warnings

-cw

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`.

Disable all warnings

-j*i

Enable all warnings

-j*w

Make all warnings errors

-j*e

Promote all warnings to errors; this option can be useful to ensure that a program compiles with no warnings whatsoever.

No error line printing

-ce

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

No error/warning limit

-q-

Never quit on any errors or warnings.

Stop on first error/warning **-q**

Undefined tag warnings **-ct**

This option enables a warning message for structure tags which are used without definition inside a 'scope'. These messages will be issued at the end of the scope in a similar manner to the warning *no reference to identifier* (93).

Error limit: *num* **-qnume**

This option controls the number of errors which the compiler will tolerate before quitting. The default is 10.

Warning limit: *numw* **-qnumw**

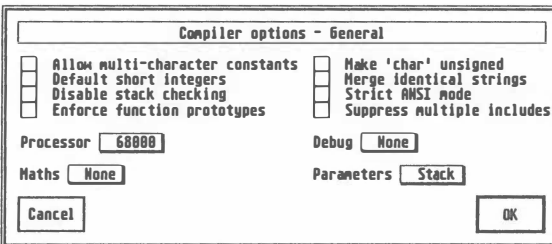
This option controls the number of warnings which the compiler will tolerate before quitting. The default is 50.

Warnings: *D* *no option*
E **-jnume**
W **-jnumw**
I **-jnumi**

This option allows control over the error messages reported by the compiler. It allows any warning to be ignored, promoted to an error, or enabled. The 4 possible actions for each message are: *D* - default action, *E* - always issue warning as an error, *W* - enable warning, *I* - ignore warning.

Compiler options - General

The General dialog includes the most common options used for selecting the compilation model. It is probably the set of options you will use most often.



Allow multi-character constants **-cm**

Default short integers **-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.

Disable stack checking **-v**

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

Enforce function prototypes **-cf**

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.

Make 'char' unsigned **-cu**

Merge identical strings **-cs**

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.

Strict ANSI mode **-ca**

Enables full ANSI compatibility mode with full diagnostics to check for portability problems. Note that a program may compile cleanly with this option in effect, however this *does not prove* that the program conforms to the ANSI standard, merely that the program will compile.

Enabling this option has the following effects:

Disables anonymous unions

Disables zero length arrays within structures (note that zero length arrays are *only* permitted within structures).

Disables long float as a synonym for double.

Causes excess (i.e. more than 2) hex digits in character constants to be discarded rather than retained.

Enforces the 'a cast does not yield an lvalue' rule.

Disallows `sizeof` and floating point numbers in pre-processor directives.

Enables trigraphs; these may be disabled via *Disable trigraph processing* (`-cg`).

Disables register keywords and new keywords. Note that some of the system header files (e.g. `dos.h`) will require you to re-enable register keywords using the *Enable '__asm' keywords* (`-cr`) option.

Disables warnings:

95 (unrecognised `#pragma` operand)
151 (use of ANSI flexible keyword ordering)

Enables warning:

148 (use of incomplete struct/union/enum tag).

Promotes the following warnings to errors:

59 (invalid storage class)
84 (redefinition of pre-processor symbol)
116 (Undefined enum tag)
101 (redundant keywords in declaration)
122 (Missing ellipsis)
132 (Extra tokens after valid preprocessor directive)
152 (cannot define function via typedef name)
162 (non-ANSI use of ellipsis punctuator)
170 (C++-style comment detected)

Forces structure equivalence to be exact type equivalence rather than member name and type equivalence.

Disallows floating point constant expressions from participating in `case` expressions.

Defines the pre-processor symbol `_ANSI` as 1. Note that this has the effect of disabling the non-ANSI features in the header files. If you require access to non-ANSI features of the header files you may disable this behaviour using the option *Compiler options - Pre-processor (#undef symbols: `_ANSI`)*.

Suppress multiple includes

-ci

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.

Processor

-m



This option used to select the target processor for which the compiler is to generate code.

68000

-m0

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.

68010

-m1

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.

68020

-m2

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 and 68040.

68030

-m3

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 and 68040.

68040

-m4

Causes the compiler to generate code optimised for the 68040 processor. In general, code for this will run on a 68020/68030 although the 68040 has instructions not found on these processors.

Any

-ma

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.

Debug

-d



This option used to select the level of debugging information generated by the compiler; either line-level for a symbolic debugger, or source level for a source level debugger.

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.

Full

-d4

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

Full/flush

-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.

Line only

-d1

Enables output of the line number /offset table.

Local

-d2

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

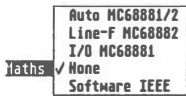
Local/flush **-d3**

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.

None **-d0**

Disables all debugging information.

Maths **-f**



This option used to select the the manner in which floating point arithmetic is performed.

Note that if you are building a desk accessory, CPX, TSR or any other form of resident/concurrent program you should only select *None* or *Software IEEE* any other option *will* conflict with the operation of any foreground programs.

Auto MC68881/2 **-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 true 68881/68882 coprocessor or an I/O based 68881 (such as Atari's SFP004 or the optional MegaST^E coprocessor) and perform floating point arithmetic on chip when possible. If no coprocessor is available then the *Software IEEE* routines will be used.

Line-F MC68882 **-f8**

Inline Motorola 68881/68882 generated instructions using the coprocessor interface. Code compiled with this option will not operate unless a 68881/68882 is installed which conforms to this interface.

I/O MC68881 **-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 or the optional MegaST^E coprocessor) and performs floating point arithmetic on chip.

None **-f**

None indicates that a program requires no floating point support. This is the default, so that those programs which use floating point must select one of the other options.

Software IEEE **-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 coprocessor if present.

Parameters **-r**



This option is used to control how the compiler is to generate subroutine calls and entries.

Both **-rb**

Defaults the compiler to use registerised parameters for all subroutine calls, yet still generate a prologue that handles both styles of parameter passing.

Stack **-rs**

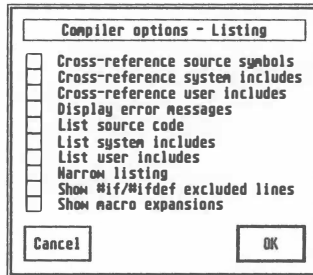
The compiler default, causes the compiler to use standard stack parameters for all subroutine calls. Those functions explicitly declared `__regargs` will use registerised parameter conventions.

Register **-rr**

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 (and the first two real items into FP0-FP1 if using *Line-F MC68882* (-f8)) for the call. Any function without a prototype or explicitly declared `__stdargs` will use the normal stack conventions.

Compiler options - Listing

This option is used to control the listing file generated by the 'big' version of the compiler.



Cross-reference source symbols

-gx

Output a cross reference of all symbols in the source file. This option is implied by the *Cross-reference system includes* (-gc) and *Cross-reference user includes* (-gd) options.

Cross-reference system includes

-gc

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.

Cross-reference user includes

-gd

Outputs a cross reference of all user-provided include files. By default these symbols are not printed.

Display error messages

-go

Output error messages to both standard out *and* the listing file. By default when generating a listing the compiler places the error messages only in the listing file.

List source code

-gs

Enables listing of the input source code.

List system includes**-gh**

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.

List user includes**-gi**

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

Narrow listing**-gn**

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.

Show #if/#ifdef excluded lines**-ge**

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

Show macro expansions**-gm**

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.

Compiler options - Object

The Object option allows fine control over the code generation of the compiler; you may well never use any of these options.

Compiler options - Object

<input type="checkbox"/> Always generate stack frames	<input type="checkbox"/> Disable stack merging
<input type="checkbox"/> Auto-load base register	<input type="checkbox"/> Long align externals
<input type="checkbox"/> Default 'far' code	<input type="checkbox"/> Long align stack
<input type="checkbox"/> Default 'far' data	<input type="checkbox"/> Optimise for space
<input type="checkbox"/> Default section names	<input type="checkbox"/> Type based stack alignment
<input type="checkbox"/> Disable auto-registerisation	

Data pointer Frame pointer

Register limits: Data Address Floating point

Names: Code: |----- Data: ----- BSS: -----

Always generate stack frame

-mf

This option forces the compiler to always generate a stack frame, even in those instances in which the frame is not required. This ensures that programs which expect to be able to 'walk' up the call chain are still able to do so.

Auto-load base register

-y

This option causes each function entry sequence to load the global data register with the value of the linker defined symbol `_LinkerDB`. Note that, in general, only the functions that will be used as entry points from an interrupt handler need to use this feature, since register A4 will be propagated by subsequent function calls, hence the `__saveds` keyword is preferable in most situations.

Default 'far' code

-r0

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.

Default 'far' data **-b0**

This option causes the compiler to change the form of addressing used to locate statics, externals and strings to less efficient full 32 bit accesses.

Default section names **-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.

Disable auto-registerisation **-mr**

Disables the automatic registerisation of variables. By default, the compiler will attempt to pick likely candidates for register variables. Note that this option has no effect if using the global optimiser.

Disable stack merging **-mc**

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.

Long align externals **-cl**

This forces alignment of all external data to longword boundaries. By default the compiler will not necessarily ensure that external data objects are placed on longword boundaries. Using this option *may* give better object code performance on full 32 bit architectures (68020/68030/68040).

Long align stack **-as**

Enable automatic longword stack realignment. By default the compiler will not necessarily ensure that the stack is longword aligned. Using this option *may* give better object code performance on full 32 bit architectures (68020/68030/68040).

Optimise for space **-ms**

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

Type based stack alignment

-aw

This option allows short and char to be passed on the stack at the appropriate size. If generating CPXs in default long integer mode this option *must* be used.

Code: *codename*

-sc=*codename*

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

Data: *dataname*

-sd=*dataname*

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

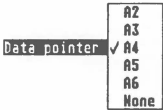
BSS: *bssname*

-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.

Data pointer: *reg*

-breg
-bn

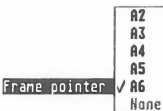


This option specifies which register the compiler is to use for its global base register (default A4), or None (-bn) if none is required (e.g. if your program is *entirely* non base-relative).

Note that use of this option will almost certainly make your program incompatible with the standard run time libraries

Frame pointer: *reg*

-rreg
-rn



This option specifies which register the compiler is to use for its frame pointer (default A6), or None (-rn) if none is required.

Note that at the time of writing Frame pointer: None (-rn) is not implemented and so may not be selected.

Register limits - Data: num

-hdnum



This option specifies the maximum number of registers used for passing char, short, int or long parameters when in -rr mode.

Note that use of this option will almost certainly make your program incompatible with the standard run time libraries.

Register limits - Address: num

-hanum



This option specifies the maximum number of registers used for passing pointer parameters when in -rr mode.

Note that use of this option will almost certainly make your program incompatible with the standard run time libraries.

Register limits - Floating point: num

-hfnum

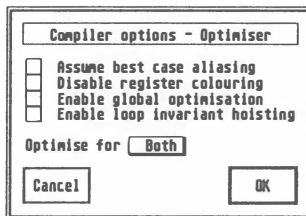


This option specifies the maximum number of registers used for passing float, double or long double parameters when in -rr and -f8 mode.

Note that use of this option will almost certainly make your program incompatible with the standard run time libraries.

Compiler options - Optimiser

The Optimiser options allow the options for the global optimiser to be selected. Note that for any of these options to have any effect you must specify the *Enable global optimisation* (-O) option.



Assume best case aliasing

-Oalias

By default the global optimiser makes decisions about how two objects overlap based on the available type information. This option disables this behaviour, allowing more optimisations, but potentially introducing unsafe optimisations.

Disable register colouring

-Onocolor

By default the global optimiser examines your code for variables with non-overlapping lifetimes and allows two or more variables to use the same physical register. Using this option disables this behaviour.

Enable global optimisation

-O

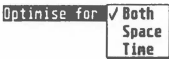
Enable loop invariant hoisting

-Oloop

When performing loop optimisation (*Optimise for - Both (-O)* or *Optimise for - Time (-Otime)*) enable hoisting of safe invariant expressions out of the loop.

Optimise for

-O



This option selects which of the optimiser's algorithms, which typically affect time and/or space, are used.

Note that the names used are misnomers; `Time` may cause slower execution, whilst `Space` may cause larger executables. The default of `Both` is almost always best.

Both

-O

This option performs both loop and very-busy expression hoisting; typically this will result in a decrease in program size *and* in execution time.

Space

-Ospace

This option performs only very-busy expression hoisting; typically this will result in a decrease in program size.

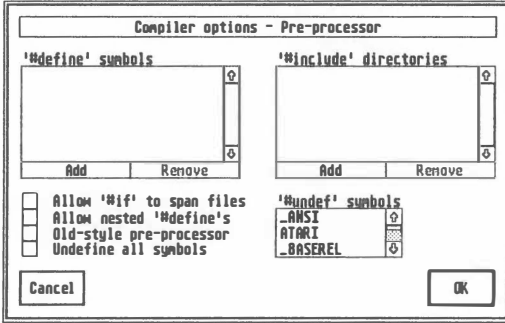
Time

-Otime

This option performs only loop optimisation; typically this will result in reduced execution time.

Compiler options - Pre-processor

The Pre-processor options allow the characteristics of the compiler pre-processor to be set up. Many of these options are somewhat esoteric and included only for backward compatibility.



Allow #if to span files

-cp

By default the compiler gives an error for pending #endifs missing at the end of #include files. This non-ANSI option suppresses this behaviour.

Allow nested '#define's

-cn

When this option is enabled, the compiler 'stacks' #define statements for the same identifier, with each #undef discarding the top element from the stack.

Old-style pre-processor

-co

This option places the pre-processor in pre-ANSI mode. In this mode, macro arguments may be substituted inside string literals (superseded by the ANSI stringisation operator, #), tokens in replacement lists may be pasted together using a comment (superseded by the ANSI token pasting operator, ##) and the #pragma title/eject/space directives may be used without the #pragma (i.e. #title/eject/space).

Undefine all symbols

-u

This option undefines all non __ prefixed preprocessor symbols which are normally pre-defined by the compiler.

'#undef' symbols: *name* **-*uname***

The #undef symbols list box shows all the pre-processor symbols which the compiler *may* predefine and which may be overridden. To disable a definition, clicking on it will 'grey' it out, clicking again will re-enable it. Note that removing symbols which would not have been generated (e.g. `_M881` when `Line-F MC68882 (-f8)` has not been set) has no effect.

'#define' symbols: *name=value* **-*dname=value***
name **-*dname***

This list box allows #define symbols to be preset. The values entered may either be of the form `name=value` to indicate a #define of the form:

#define name value

or simply name to indicate

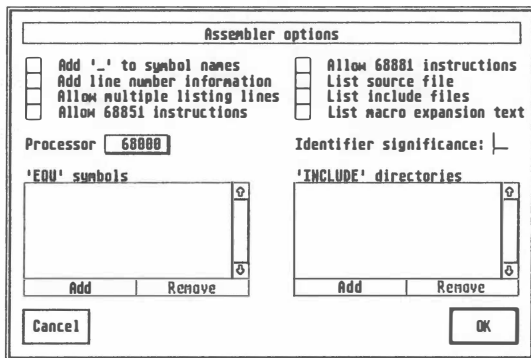
#define name

'#include' directories: *dir* **-*idir***

This list box allows a set of directories which the compiler is to look in for #include files to be set up. Note that although the list box allows an infinite number of entries, the compiler only permits 16.

Assembler options

The Assembler options item are the main place (for a project or single file) in which assembly options are set.



Add ' _ ' to symbol names **-u**

This option automatically prefixes all external references with an `_`. If references to C labels have already been prefixed with an underscore in the source, the option is not needed.

Add line number information **-d**

This option activates the debugging mode (in the same way as the compiler *Debug - Line only* (-d1) option)

Allow multiple listing lines **-lm**

List additional data generated for source lines which cannot be accommodated alongside the original source line (i.e. allows multiple listing lines for each source line). This option implies the *List source file* (-l) option.

Allow 68851 instructions **-m9**

Allow 68881 instructions **-m8**

List source file **-l**

This option causes a listing of the source file to be written to the file *source.lst*. The listing displays the appropriate program counter and code information alongside the assembly source.

List include files **-li**

List the source for text from INCLUDE files as well as the original source file. This option implies the *List source file* (-l) option.

List macro expansion text **-lx**

In addition to listing the call to a macro this option causes the expansion text from macros to be listed. This option implies the *List source file* (-l) option.

Processor

-m



This option controls whether warnings are generated when code for the relevant processor is encountered; in general each processor provides a superset of the instructions of its predecessor.

68000	-m0
68010	-m1
68020	-m2
68030	-m3
68040	-m4
68332	-m32

Note that for processors with built in FPU's or MMU's then the relevant subset of 'co-processor' instructions are also enabled without the need to specify the *Allow 68851 instructions (-m9)* or *Allow 68881 instructions (-m8)* options

Identifier significance: sig

-nsig

This option specifies the number of characters the assembler is to retain for identifiers. This can be useful if more than the default of 31 is required, or to reduce to 7 or 8 for compatibility with very old programs.

'EQU' symbols: name=value
name

-dname=value
-dname

This list box allows EQU symbols to be preset. The values entered may either be of the form name=value to indicate an EQU directive of the form:

name EQU value

or simply name to indicate:

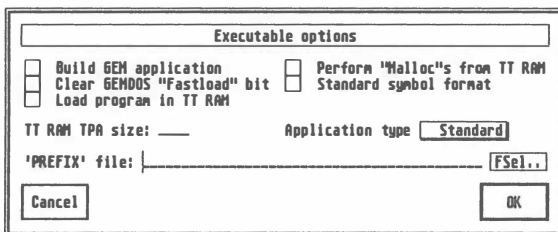
name EQU 1

'INCLUDE' directories: *dir*

-idir

This list box allows a set of directories which the assembler is to look in for INCLUDE files to be set up. Note that although the list box allows an infinite number of entries, the assembler only permits 16.

Executable options



Build GEM application

-Lg

The *Build GEM application* (-Lg) option tells the compiler that you intend to build a GEM program and that it should link with the libraries providing the GEM facilities. The integrated compiler also uses this setting to determine whether a program which you are working on should be run as a GEM or a TOS program on selecting Run "... " or Debug "... " from the Project menu.

Clear GEMDOS "Fastload" bit

NOFASTLOAD

This disables the setting of the "Fastload" bit in the program header of an executable program. This means that the whole of the TPA will be zeroed rather than just the BSS section.

Load program in TT RAM

TTLOAD

This option sets the appropriate bit in an executable program's header to indicate the the application would prefer to load into TT RAM if available.

Perform "Malloc"s from TT RAM

TTMALLOC

This option sets the appropriate bit in an executable program's header to indicate the the application would prefer to have GEMDOS Malloc's satisfied from TT RAM if available.

Standard symbol format

DRISYM

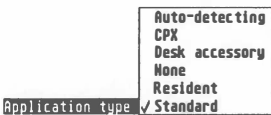
When generating symbols in an executable program, the linker normally generates them in 'HiSoft extended format'; this has the advantage of permitting 22 characters of significance, compared to 8 for standard DRI symbols. Many debuggers now understand the HiSoft format, however this option may be used for backward compatibility.

TT RAM TPA size: *size*

TPASIZE *size*

Sets the size of TPA required for loading into alternative RAM. This value sets the minimum amount of alternative RAM, in Kbytes, which must be free for a program which has the TTLOAD bit set. The minimum value is 128, the maximum 2048 (2Mb). Note that this option implies the *Load program in TT RAM* (TTLOAD) option.

Application type



The Application type options allow you to tell the compiler which sort of application you are building.

Auto-detecting

-ta

This option forces the use of the automatic program type detection code, allowing an application to determine whether it is operating as a normal program, from the auto folder or as a Desk accessory *at run-time*. The external variable `_XMODE` can be used to determine the current mode.

CPX

-tx

This option forces the program to be built for CPX operation. A CPX is designed to be used in conjunction with Atari's XControl Desk accessory and so must be tested in that environment. Note also that *every CPX must have* a PREFIX file specified, as built with CPXBUILD.

Desk accessory

-ta

This option builds the program as a Desk accessory. A Desk accessory is designed to be loaded directly by the Desktop and so must be tested in that environment.

None

-t=

This option sets the compiler to link no start up code to the modules in the project. This can be useful for building applications which either require no startup code (because they are designed that way) or for implementing application types not directly available from this menu (e.g. Harlekin HPG modules).

Resident

-tr

This option forces the use of the resident program startup code. The use of this application type allows the resulting application to be made resident using the Tools - Resident option.

Standard

no option

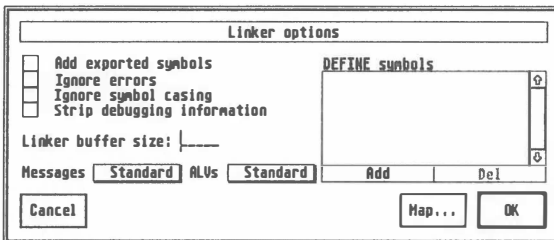
This is the default application type and indicates that the standard C startup code should be used. This is by far the most common option.

'PREFIX' file: *file*

PREFIX *file*

This specifies a file which is to be prepended to the output file; this is particularly useful for building control panel extensions.

Linker options



Add exported symbols

ADDSYM

This option causes the linker to discard external symbol information transmitted from the compiler or assembler as a result of a debugging option and replacing it with a symbol table constructed from global symbol definitions. This has the advantage that library names then appear in the symbol table, however any non-global symbols disappear.

Ignore errors

IGNORE

Force the linker to continue after serious errors. Note that the use of this option may result in a non-executable file if an error occurs.

Ignore symbol casing

NOCASE

Make the linker ignore the casing of symbols whilst resolving external references and definitions.

Strip debugging information

NODEBUG

This option strips any debugging information from the input files which was generated as a result of a compiler or assembler debugging option. Note that if memory is short enabling this option will allow the link to take place in less memory.

Linker buffer size: size

BUFSIZE size

This option sets the linker I/O buffer size. By default, all I/O is done in blocks as large as the available memory permits; this leads to extremely fast link times. This option may be useful if so little memory is available that the normal allocation scheme fails due to lack of memory.

Messages



The linker allows the level of messages generated to be set using this option.

Quiet

QUIET

Causes CLink to print no messages at all unless an error occurs.

Standard

no option

This is the default option in which the linker prints sign on, sign off and brief summary messages.

Verbose

VERBOSE

Causes CLink to print out the name of each file as it processes it and a summary of memory usage and elapsed time on completion.

ALVs



The ALVs option allows you to customise the manner in which the linker generates ALVs.

When the linker is collecting the CODE type sections together, if any are more than 32K apart and a 16-bit PC relative access is attempted, rather than simply fail with an out-of-range error message, CLink redirects the access to a JMP to the same location.

This jump is known as an *automatic link vector* or *ALV*. Note that this may cause problems if you attempt to access data using PC-relative mode, although this is not recommended anyway since on the 68030 there are separate code and data caches which can cause consistency problems.

Inhibit

XNOALVS

Prevents CLink from creating ALVs to resolve 16 bit PC-relative code. Note that the use of this option may force CLink to fail in pass 2 with a fatal error.

Standard

no option

This is the default option; it is unlikely you will ever want to use anything other than this option.

Warnings

NOALVS

Forces CLink to warn you when it creates ALVs to resolve 16 bit PC relative code. This can be used to watch for CLink creating a non-relocatable object from what was intended to be relocatable code.

DEFINE symbols: *name=value*
name=name

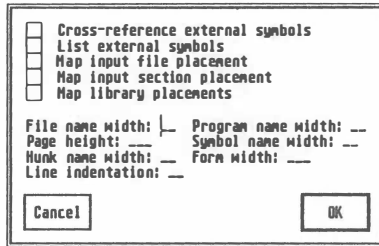
DEFINE *name=value*
DEFINE *name=name*

This list box allows linker DEFINE symbols to be preset. The values entered may either be of the form *name=value* to assign a specific value or *name=name*, to indicate an alias for another external label.

This option is particularly useful in conjunction with the PRELINK option to force certain routines to be pulled from the library even though no references to them exist.

Map...

A map file is a file describing the order and location of files and variables processed by the linker for perusal by the user.



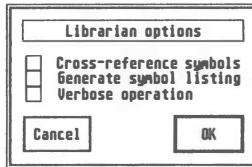
Cross reference external symbols	MAP...X
List external symbols	MAP...S
Map input file placements	MAP...F
Map input section placements	MAP...H
Map library file placements	MAP...L

These options control which parts of the map file are generated.

File name width: width	FWIDTH width (16)
Page height: height	HEIGHT height (55)
Hunk name width: width	HWIDTH width (8)
Line indentation: indent	INDENT indent (0)
Program name width: width	PWIDTH width (8)
Symbol name width: width	SWIDTH width (8)
Form width: width	WIDTH width (80)

These options are used to customise the layout of the map file; the default values are shown in parentheses.

Librarian options



Cross-reference symbols -X

This option causes a librarian listing to be generated and includes a cross reference of all symbols.

Generate symbol listing -S

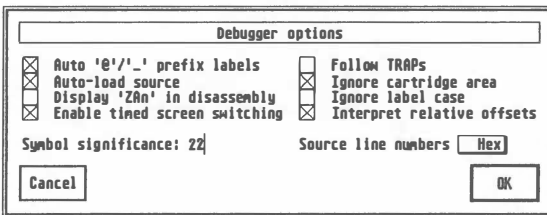
This option causes a librarian listing to be generated and includes a listing of all public symbols defined in the module .

Verbose operation -V

Forces the verbose mode of operation; in this mode the librarian prints out its progress whilst running.

Debugger options

The integrated compiler automatically makes available one of the medium level debuggers, MonSTC or MonTTC depending on which machine type it detects. The options for this debugger (set via Ctrl-P inside Mon) may also be set up within the environment:



If you wish to use an alternative debugger (e.g. DB from Atari) this can be done by naming a copy of the debugger MonSTC or MonTTC respectively.

The integrated compiler will notice such uses and not pass the debugger strange options! Note that you should only make MonSTC or MonTTC resident (using the Resident configuration option), attempting to make other debuggers resident will almost certainly crash the machine.

Auto '@/'_ ' prefix labels

With this option set Mon will try prefixing symbols by `_` and `@` if it cannot find a label, so that if you enter `main` and there is no label called `main`, then Mon will try `_main` or if this doesn't exist then it will try `@main`.

Auto-load source

Using the default settings, Mon will automatically load a C source file and run your program until the label `_main`, (i.e. the beginning of your function `main`), ready for you to set a breakpoint in the code. Mon loads the source file corresponding to the first module with debug information in the file that you are debugging.

Display 'ZAn' in disassembly

This option allows advanced programmers to enable the display of the normally hidden Z registers used by some 680x0 instructions.

Note that the Display 'ZAn' in disassembly option will be disabled if running on an ST.

Enable timed screen switching

Defaulting to `On`, this causes the display to switch to that of your program only after 20 milliseconds. It should be switched off when a program is about to change a screen's address or resolution and then turned back on afterwards.

Follow TRAPs

By default, single-stepping and the various forms of the Run command treat TRAPs, Line-A and Line-F calls as single instructions. However by turning this option `On` the relevant routines will be entered allowing ROM code to be investigated.

Ignore cartridge area

When this option is selected the Find command will not search the ROM cartridge area of the memory map. You should select this if you have hardware other than a ROM in this slot.

Ignore label case

This option defaults to Off. If it is set to On then if you enter fred in an expression the subsequent search will give the value of the first symbol that matches this, ignoring case, thus finding FRED, fred or FrEd. This option is useful for lazy typists who use the same name with different casing.

Interpret relative offsets

This option defaults to On and affects the disassembly of the address register indirect with offset addressing modes, i.e. xxx(An). With the option on, the current value of the given address register is added to the offset and then searched for in the symbol table. If found it is disassembled as symbol(An). This option is required to show the addresses of your global variables if they are accessed via an address register.

Symbol significance: sig

This option specifies the number of characters the debugger treats as significant for identifiers. This can be useful if less than the default of 22 is required.

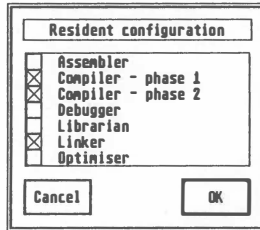
Source line numbers



MON can either show line numbers in your source window in decimal, hex or not at all.

Resident configuration

The Resident option allows the selection of which tools are resident in memory. The defaults are as shown below. Note that a tool which is not resident will be loaded when needed by the project manager.



If you find that during compilation the compiler quits with an out of memory error, removing some of the resident tools may well ease the situation, at the slight expense of a longer initial compilation cycle.

LC1, LC2, GO

The Compiler

The Lattice C 5.50 compiler has undergone many changes to bring the compiler into line with the language definitions of the ANSI standard. As such the pre-processor symbol `__STDC__` is now always set to 1, even when `-ca` is not set, giving an ANSI model, with extensions. If `-ca` is specified then `_ANSI` will be set to 1. Note that the compiler has undergone no formal validation or independent testing, although we have a high degree of confidence in its ANSI compliance.

New Language Features

ANSI compliance

Extern scoping model

The compiler deals with implicit and explicit in-block (cf. global) extern declarations according to ANSI, i.e. their scope is restricted to that of the block. The *Make 'extern' declarations global* (`-cx`) option can be used to force their scope to global for compatibility with pre-5.50 and other non-ANSI compilers.

Note that the meaning of the `-cx` option has changed; the functionality of the old `-cx` option is still available via `-x`.

This change has some surprising side effects:

```
void fn(long);

void f(void)
{
    extern void fn();

    fn(42);
}
```

results in `fn()` being called without reference to the prototype.

Flexible keyword ordering

ANSI flexible keyword ordering is fully supported. This allows you to write such obfuscated declarations as:

```
int long unsigned typedef size_t;
```

any such abuses are flagged by the compiler via the warning use of ANSI flexible keyword ordering (151).

The warning is disabled by default in ANSI mode. Note that the placement of the storage class specifier (`typedef` in this instance) is marked as an obsolescent feature even by ANSI.

Float as single

The compiler now handles expressions of type `float` in single precision; this is to conform to the requirements of the ANSI standard.

Note that in general the use of `float` as a computational type is discouraged unless using one of the co-processor math options.

Redundant keyword combinations

Redundant keyword combinations which are detected generate the warning redundant keywords in declaration (101) in non-ANSI mode; in ANSI mode this warning is automatically promoted to an error. Redundant keyword situations include:

```
typedef volatile int mytype;
...
return (volatile mytype)10; /* volatile is redundant */
```

Ref/def model

The ANSI external data reference/definition model is now strictly enforced. This makes programs, of the form shown below, illegal:

```
static int i1;
int i1 = 42;
```

Restriction of register arrays/aggregates

register arrays/aggregates are restricted in their use to those sanctioned by ANSI. The only permitted use for such a register array is `sizeof()`.

Scoping rules for 'no-linkage' objects

The compiler restricts typedefs to exactly one instance in a single scope. This is to comply with the rule "If an identifier has no linkage, there shall be no more than one declaration of the identifier".

Trigraphs

Trigraphs are now fully implemented. They are disabled by default, or enabled in ANSI mode. Due to the overhead of parsing these sequences using the compiler with them enabled is approximately 5% slower, hence we recommend that if using ANSI mode you disable them using the *Disable trigraph processing* (-cg) option.

Type composition of scoped declarations

Scoped declarations have their types correctly composed as per ANSI rules.

typedef model

The exact ANSI typedef model is now used; this allows confusing usages such as:

```
typedef int t;

struct t {
    unsigned t:5;    // unsigned bit field named t
    const t:5;      // unnamed const int bitfield
};
```

Valid storage classes of local functions

Local functions declared using any storage class other than static will elicit a warning in non-ANSI mode, or an error in ANSI mode.

C++ features

Various features have been 'stolen' from C++ which were omitted by the ANSI committee. All are disabled when the `-ca` option is used.

Comments

C++ style comments are now permitted. The normal `/...\
used.`

Ellipsis

The C++ style for variable argument functions is now available, incurring the warning *non-ANSI use of ellipsis punctuator* (162).

Anonymous unions

The outer tag of unions may be omitted, since this is usually only a placeholder anyway. For example:

```
struct node {
    struct node *next;
    int type;
    union {
        short sval;
        long lval;
        float fval;
        double dval;
    };
};
```

Floating point `__asm` support

Functions declared using the `__asm` directive may now be passed floating point registers viz:

```
void __asm fp(register __fp1 double);
```

__interrupt keyword

The `__interrupt` keyword has always been implemented, but caused no significant change to the run time model. In the 5.50 release the `__interrupt` keyword modifies the function entry sequence such that it is as if a structure of type `struct except` (from `sys/except.h`) has been passed to the function. In addition stack checks are automatically disabled for the function, whilst an RTE is used for exit. Note that the formal parameter *must* be declared `volatile` if you intend to modify any part of it prior to returning, i.e.

```
void ex (struct except volatile x)
{
    x.mc680x0.f10.ssr &= ~0x100; // clear 68030 rerun flag
}
```

Note that this facility is for the advanced user; you *must* have a good working knowledge of the processor and its exception structure in order to use it. For details of `struct except` you should see the header file `sys/except.h`.

ANSI relaxations

Modifiable lvalues

The construct:

```
*((long *)bar)++=100;
```

is now permitted in non-ANSI mode. This is a relaxation of ANSI which requires that casts do not form lvalues. This causes `bar` to be incremented, as expected, by `sizeof(long)` rather than `sizeof(*bar)`.

Signed and sized bit fields

signed and explicitly sized bitfields (`char/short/long`) are now permitted. The default for all such types is unsigned (ANSI leaves this as implementation defined). Note that to enable the explicit sizing feature you must specify the *Allow explicitly-sized bitfields* (-cb) option.

Zero length arrays

The compiler now permits zero length arrays embedded within structures when in non-ANSI mode. This is a common trick used for allocating a variable length structure; its use is *explicitly* prohibited by ANSI. Typical uses are of the form:

```
struct name {
    int length;      // length of username
    char name[0];   // characters of username
} user;
```

Listing control directives

#pragma eject

The `#pragma eject` directive causes the remainder of the listing page to be left blank and a new page started.

If the *Old-style pre-processor* (`-CO`) option has been specified then directive may be given as

```
#eject
```

#pragma space lines

The `#pragma space` directive causes the remainder of the listing page to be left blank and a new page started if there are fewer than `lines` lines remaining on the current page. Note that `lines` may be a general pre-processor expression.

If the *Old-style pre-processor* (`-CO`) option has been specified then directive may be given as

```
#space ...
```

#pragma title title

The `#pragma title` directive sets the title printed at the top of subsequent pages to `title`.

If the *Old-style pre-processor* (-c0) option has been specified then directive may be given as

```
#title ...
```

Error control directives

#pragma error num

The `#pragma error` directive is used to promote the message `num` to an error; `num` may be any general pre-processor expression.

Note that a warning which has been promoted to an error *may not* be demoted to a warning again.

#pragma ignore num

The `#pragma ignore` directive is used to indicate that the warning message `num` should be ignored; `num` may be any general pre-processor expression.

#pragma warning num

The `#pragma warning` directive is used to indicate that the warning message `num` should be enabled; `num` may be any general pre-processor expression.

New error/warning messages

Warning 88, argument type incorrect, has been refined to be either 171 (implicit cast of integral argument), 135 (assignment to shorter data type (precision may be lost)), or the existing 85. This gives a much better indication of potential problems.

Warning 103 (uninitialised constant) is now issued for global variables in addition to local variables.

Warning 122, Missing ellipsis, and warning 132, Extra tokens after valid preprocessor directive are now always enabled, previously they were auto-enabled in ANSI mode; in ANSI mode they are promoted to error status.

.

There are several new error/warning messages:

register __fpX requires -f8 switch on LCI

A function of the form:

```
__asm fn(register __fp0 double);
```

has been defined/used without the -f8 flag having been specified.

146 (W) long case value in short switch

The compiler has detected a switch value whose range exceeds the range of the switch type. The compilation will continue using the truncated value.

148 (W) use of incomplete struct/union/enum tag <name>

The named struct/union/enum tag had been used without a corresponding in scope definition. This warning is normally disabled, or enabled in ANSI mode.

149 (W) undefined struct/union/enum tag in prototype scope

An undefined struct/union/enum tag has been encountered within a prototype. Because a prototype forms its own scope it is thus impossible to have any type (within the translation unit) which is compatible with it.

This warning is normally only issued for enums when in non-ANSI mode, or for all types in ANSI mode. This is because the compiler uses the ANSI cross-translation-unit model (member name and type equivalence) for structure equivalence when in non-ANSI mode, rather than exact type equivalence in ANSI mode.

151 (W) use of ANSI flexible keyword ordering

The compiler has detected a use of flexible keyword ordering in declarations, as permitted by ANSI. Note that use of such orderings is generally confusing and less portable to older compilers. This message is disabled by default in ANSI mode (-ca).

152 (W) cannot define function via typedef name

An attempt is being made to define a function via a typedef name. In default mode this is accepted with this warning; in ANSI mode (-ca) it constitutes an error.

153 (W) use of string constant concatenation

The compiler has detected a use of ANSI string concatenation. This warning is normally disabled.

159 (W) use of unary minus on unsigned value

The compiler has detected a unary minus on an unsigned expression; often this will not indicate an error, although it can be useful for tracking down unexpected effects.

161 (W) no prototype at definition of public function

At the definition of a public function (i.e. `non-static`) there was no in scope prototype. This may indicate a prototype missing from a global header file. This warning is normally disabled.

162 (W) non-ANSI use of ellipsis punctuator

The compiler has detected a non-ANSI usage of the ellipsis punctuator, typically this will indicate that C++ syntax was used, i.e.

```
void fn(const char *s ...);
```

This warning is promoted to an error in ANSI mode.

166 unbalanced comment

The end of file was reached whilst a closing comment was still outstanding. Note that this error is a refinement of the previously all encompassing error, unexpected end of file.

167 (W) nested comment detected

This warning is issued whenever the compiler detects an apparent use of comment nesting. Note that this may indicate a portability problem to compilers which do not allow the user to dictate whether comments nest or not. This warning is normally disabled.

170 (W) C++-style comment detected

The compiler has detected a usage of the C++ `//` style commenting. This warning is normally disabled and an error in ANSI mode.

The argument to a function, or function return value, has been implicitly cast from one integer type to another; note that this may only occur if there is an in-scope prototype. This is a refinement of warning 88, argument type incorrect, and is disabled by default.

Pre-processor symbols

This is a current list of pre-processor symbols which the compiler predefines:

Optional definitions

Name	Option
<code>_ANSI</code>	<code>-ca</code>
<code>_BASEREL</code>	<code>-b1</code>
<code>_DEBUG</code>	<code>-d1..-d5</code>
<code>LPTR</code>	without <code>-w</code>
<code>_M881</code>	<code>-f8</code>
<code>_MDOUBL</code>	<code>-fd</code>
<code>_MLATTICE</code>	<code>-f1</code>
<code>_MMIXED</code>	<code>-fm</code>
<code>_MSINGLE</code>	<code>-fs</code>
<code>__PLAIN_CHAR_UNSIGNED</code>	<code>-cu</code>
<code>_PCREL</code>	<code>-r1</code>
<code>_REGARGS</code>	<code>-rr</code>
<code>_SHORTINT</code>	<code>-w</code>
<code>SPTR</code>	<code>-w</code>
<code>_UNSIGNEDCHAR</code>	<code>-cu</code>

Static definitions

These definitions are always made by the compiler, regardless of the options, unless overridden using the *#undef symbols* (-u) option.

Name	Value	Meaning
ATARI	1	Host Machine
LATTICE	1	Compiler Name
LATTICE_50	1	Compiler Version
LATTICE_550	1	Current compiler release
M68000	1	Processor type

Permanent definitions

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__	50	Current minor version number.
__STDC__	1	ANSI operation mode
__TIME__	"time"	Time at which compilation was started
__VERSION__	5	Current major version number.

Changes to the run-time model

There are a number of changes to the run time model which will require most programs to be fully recompiled, whilst those with assembly language portions may need rewriting. This section attempts to cover those points which may cause problems within assembler portions.

We strongly recommend that *all* applications are recompiled in their entirety.

Register passing mode

In registerised parameter passing mode (`-rr` or `__regargs`) changes have been made to the way parameters are passed. In true 68882 mode (`-f8`), FP0 and FP1 are used to pass the first two double parameters to the function in emulation mode (`-f1`) if the first parameter in a function is of type double then registers D0/D1 are used to pass the parameter (strictly if no integer parameters precede them that would have used D0/D1). Note that this change alone means that *any* code using real maths and `-rr` *must* be recompiled.

__savesd and stack checks

Functions which are declared `__savesd` now automatically disable stack checking on a per-function basis. Note that this *cannot* apply to functions called by the `__savesd` function.

A2 as a register variable

The compiler now makes register A2 available as a register variable, giving 3 pointer type register variables: A2, A3 & A5. If this register is not preserved across calls then your program is almost certain to crash; previously the compiler rarely relied on A2 across a subroutine call.

__asm functions

Functions declared `__asm` are now always `_` prefixed at link time. This means that `@` is reserved for `__regargs`. This also means that it is no longer legal to write:

```
__asm __stdargs fn(...);
```

Note that for functions declared `__asm` it is quite legal to omit a register specification for a parameter; any such parameters will be passed on the stack in the normal way. Such usages elicit a harmless warning however.

Signed and sized bit fields

Changes to bitfields now mean that in default short integer mode (-w) a bitfield is now shorter than in the pre-5.50 releases. If you have used bitfields at all you should consider a *complete* recompile.

LC.TTP

With the release of Lattice C 5.50 the functionality of the integrated compiler has been greatly improved, however for those die-hard command line or dedicated editor users, the capabilities of the command line compiler driver have also been extended.

New LC.TTP driver options

- + list the options passed to each phase of the compiler; prior to starting compilation the LC.TTP lists the options which it is going to pass to each phase of the compiler.
- E automatically invoke \$EDITOR on error. When this option is used, the -j option is automatically turned on (to generate an error file) and the editor given by the EDITOR environment variable started with the command line:

```
$EDITOR [options] <error-file> <source-file>
```

the options part of this command line may be set by appending an '=' to the -E option, together with any options required. For example, MicroEMACS (*not* supplied) allows the startup file `error.cmd` (supplied on Disk 3) to be automatically run when passed the -e option, hence setting up your environment variables as:

```
EDITOR=c:\bin\ue.ttp ; or whatever  
LC_OPT=-E=-e
```

would automatically invoke MicroEMACS when errors are reported, together with a script loaded to parse the errors.

Note that the -E option will normally delete the errors file after any editor has been successfully called.

- g= specify listing file name. By default any listing file is sent to the source file name, but with the extension .LST. This option allows this to be changed if required.

- j generate error file. By default any listing file is sent to quad file directory, but with the extension `.ERR`, this may be changed by specifying the option as `-j=filename`.
- 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 `ADDSYM` 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.
- c This invokes the `NOCASE` option of the linker. It forces case-insensitive 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.
- i This letter directs the linker to ignore errors during linking; it is equivalent to the `IGNORE` linker keyword.
- 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 `lvg.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.

At the end of the `-L` options, if an `'='` is present then the remaining part of the `-L` option specifies the output file name.

- o The exact meaning of `-o` has changed such that it is now the name of the compilation output file, i.e. if pre-processing or precompiling (processes which do not require `LC2`) then the output filename is now specified by `-o`, rather than `-q` as in pre 5.50 releases.
 - N no compile, link named files only. This is useful to allow linking of all files on the command line with no compilation taking place what-so-ever.
 - S This option specifies the stack size for any or all compiler phases. Because the compiler uses some recursive algorithms, very complex expressions may cause it to run out of stack space. If this happens, you can increase the stack beyond its 16K default size in the following way:
 - S=n Specifies the stack size for phase 1, phase 2, and the optimiser.
 - S1=n Specifies the stack size for phase 1.
 - S2=n Specifies the stack size for phase 2.
 - S0=n Specifies the stack size for the optimiser.
- The value `n` in the preceding list is the number of bytes in the stack. For example, you can specify 16 kilobytes as `16384`, `16k` or `16K`.
- tx use `.CPX` startup stub
 - tx=y use `.CPX` startup stub, specifying file `y` as the `CPX` header to be prepended to the executable.
 - Y syntax check only (`-y` to `lc1.ttp`).
 - z2 generate DRI format object code

The `LC_OPT` variable can have the form `LC_OPT="FILE"`. In this instance `lc.ttp` reads the options from the named file.

New compiler options

In addition to the new LC.TTP driver options there are many additional compiler options. All of these are listed in the Integrated compiler section under their relative subsection. The format of these entries is as follows:

Assume best case aliasing **-Oalias**

The *Assume best case aliasing* option is enabled via the command line option **-Oalias**, the emboldening of the entire option indicates that the text should be typed literally.

Pre-processor expansion buffer: size **-zsize**

The *Pre-processor expansion buffer* option is enabled via the command line option **-zsize**, the non-bold size part indicates that the user preference is entered here (e.g. **-z10000**).

Ignore symbol casing **NOCASE**

This is a linker option (indicated by the lack of a preceding minus); there is no way of passing these directly from LC.TTP to CLink, however many of these options have -L equivalents (-LC in this instance). The full linker commands may either be passed on the command line to CLink, or via a WITH file.

Linker

To better support new Atari machines (e.g. the TT), several new options have been added to the linker, in addition to greatly improved speed for those with plenty of memory.

The new (or modified options are):

ADDSYM	Replace symbol table with table built from exported symbols.
BUFSIZE size	If size > 0 set input <i>and</i> output buffer size to size bytes, if size < 0 set output buffer size to -size bytes. By default the linker now buffers the whole of input source files for as long as possible, this often means that no re-reading is necessary for the second pass, although it may run out of memory as a result. If this happens, try setting an buffer size of 1024 to try and release more memory. If you have plenty of memory you may like to increase the output buffer from the default of 4K, by specifying an output buffer size of say -32K.
DRISYM	Force symbols placed in the executable to be of standard format. Note that this option is only effective when generating an executable file.
NOCASE	Ignore casing of symbols whilst resolving externals
NOFASTLOAD	This disables the setting of the 'fast load' bit in the program header of an executable program. This means that the whole of the TPA will be zeroed rather than just the BSS section. Note that this option is only effective when generating an executable file.
PREFIX file	This specifies a file which is to be prepended to the output file; this is particularly useful for building control panel extensions. Note that this option is only effective when generating an executable file.

TPASIZE n	Sets the size of TPA required for loading into alternative RAM. This value sets the minimum amount of alternative RAM, in Kbytes, which must be free for a program which has the TTLOAD bit set. The minimum value is 128, the maximum 2048 (2Mb). Note that this option automatically enables the TTLOAD option. Note that this option is only effective when generating an executable file.
TTLOAD	This sets the load into alternative RAM bit in the program header of an executable program. Note that this option is only effective when generating an executable file.
TTMALLOC	This sets the malloc-from alternative RAM bit in the program header of an executable program. Note that this option is only effective when generating an executable file.

The XADDSYM option has been removed, the default is now extended format symbols, also the way in which symbols are generated has changed... Within a linkable file both the assembler and compiler generate a HUNK_SYMBOL section (this contains the values for all debugging symbols), when any of the relevant debug options are enabled.

The semantics of the ADDSYM option have also therefore changed... If present this option causes the contents of all HUNK_SYMBOL sections to be discarded and the executable symbol table built from the exported symbols. This has the advantage that library names then appear in the symbol table, however any non-global symbols disappear.

CLINKWITH; the CLink environment variable

The environment variable CLINKWITH, if available, is taken by CLink to be the name of a WITH file whose contents is to be searched *before* any of the other files mentioned on the command line. This allows a template WITH file to be generated with the standard startup and library files mentioned in the CLINKWITH file, whilst the additional files are specified on the command line. The format of the CLINKWITH variable should be:

```
CLINKWITH=c:\lattice\default.lnk
```


Note that if you are running on a TT you usually want the load bits set to run in TT RAM etc., but CLink defaults to TTLOAD etc. off, for compatability. If a CLINKWITH file is specified and includes the lines:

```
TTLOAD  
TTMALLOC
```

programs will automatically be linked to go into TT RAM.

ASM

The Assembler

The Lattice Macro Assembler supports the development of assembly language modules for use with C programs. Because the Lattice C Compiler generally produces very good machine code you seldom have to resort to assembly language programming. However, some intimate relations between hardware and software are best achieved in the assembly language environment. Also, assembly language is sometimes necessary when you want to get the best combination of code size and speed.

The assembler handles the complete set of Motorola 680x0 instruction mnemonics as well as an extensive set of assembler directives and a powerful macro facility. It can, therefore, be used to develop complete systems in assembly language. Nonetheless, it is provided primarily to supplement the C compiler and has not really been designed for large assembly language projects. For such tasks a full assembler package, such as DevpacST should be used giving more power for the assembly language programmer.

Basic concepts

The assembler reads a source file and produces an object file in the Lattice object file format, along with an optional listing of the source and assembled code. The source file is assumed to have a `.S` extension and the object file is produced with a `.O` extension.

Source format

Each assembly language source line has the following format:

label operation operands comment

White space (i.e. spaces and tabs) can appear before any field and must appear between the operation and operand.

The four fields of the source line are described below:

Label

The `label` field is optional. If it is present and is preceded by white space, it must be followed immediately by a colon. That is how the assembler determines that the field is a label and not an operation. If there is no white space before the label, then the colon may be omitted.

A label can normally be up to 63 characters long and can contain letters, digits, underscores, periods, at symbols (@) and dollar signs. It cannot start with a digit, and the case of letters *is* significant. For example, labels `XYZ`, `xYZ`, and `XyZ` are distinct.

Local labels are supported using the Motorola standard syntax of a decimal number followed by a dollar character. They may be used between two non-local labels and need only have unique names within that scope. Note that unlike `GENST`, starting a label with a period *does not* signify a local label.

Operation

The `operation` field contains the name of an instruction, assembly directive, or macro. This field may not begin a line; if no label is present, then the line must begin with white space. If a label is present but is not followed by a colon, then white space must separate the label and operation fields.

The case of this field is *not* significant. That is, operation `MOVE` is the same as `move`, this applies equally to macros.

Operands

The `operands` field contains zero or more expressions, depending on the particular operation. For some operations, the `operands` field is optional or never used. Expressions are composed of constants, variables, and operators.

A *constant* is a decimal, hexadecimal, octal, or binary number. The default number base is decimal, and the other bases are indicated by a prefix:

Number representations

Number	Representation	Example
Decimal	a string of decimal digits	1234
Hexadecimal	\$ followed by a string of hex digits	\$89AB
Octal	@ followed by a string of octal digits	@743
Binary	% followed by zeros and ones	%10110111
ASCII Literal	Up to 4 ASCII characters within quotes	"AC9T"

A *variable* is a label name or a name defined via an assembler directive. The special variable, * (asterisk) can be used to signify the current program counter.

An *operator* is one of the following:

Order	Operator	Meaning
1	-	Unary minus
	~	Bitwise NOT
2	<<	Left shift
	>>	Right shift
3	&	Bitwise AND
	!	Bitwise OR
4	*	Multiply
	/	Divide
	%	Modulo
5	==	Equal to
	!=	Not equal to
	<	Less than
	<=	Less than or equal to
	>	Greater than
	>=	Greater than or equal to
	+	Add
-	Subtract	
6	^	Bitwise Exclusive OR

The Order column indicates the order in which operators are processed. Operators of the same precedence are processed from left to right. For example, in the expression

ABC+DEF* -PDQ

the negation of PDQ is performed first, followed by the multiplication and then the addition, although this can be overridden by the use of parentheses as in,

```
(ABC+DEF)*-PDQ
```

Each expression represents a 32-bit value. An *absolute expression* is one that contains only constants (literal or equated), while a *relocatable expression* contains symbols whose value is determined during linking.

Comment

This field is any text appearing after an operation, associated operands and white space. A comment may also be specified after a label or on a blank line when prefixed with a semi-colon or asterisk.

Addressing modes

The addressing modes supported by the Lattice assembler are as follows:

Mode	Example
Dn	add.w d1,d0
An	addq.w #1,a1
(An)	add.w (a1),d0
(An)+	add.w (a1)+,d0
-(An)	add.w -(a1),d0
d16(An)	add.w 10(a1),d0
d8(An,Xn)	add.w 10(a1,a2.1),d0
bd(An,Xn)	add.w \$10000(a1,a2.1),d0
([bd,An],Xn,od)	add.w ([10,a1],a2.1,20),d0
([bd,An,Xn],od)	add.w ([10,a1,a2.1],20),d0
(xxx).W	add.w (100).w,d0
(xxx).L	add.l (100).l,d0
#<data>	add.l #100,d0
d16(pc)	add.w 10(pc),d0
d8(pc,Xn)	add.w 10(pc,a2.1),d0
bd(pc,Xn)	add.w \$10000(pc,a2.1)
([bd,pc],Xn,od)	add.w ([10,pc],a2.1,20),d0
([bd,pc,Xn],od)	add.w ([10,pc,a2.1],20),d0

where:

d8	8 bit number
d16	16 bit number
bd	32 bit byte displacement
od	32 bit outer displacement
An	Address register (a0-a7)
Dn	Data register (d0-d7)
Xn	Index register (d0-d7 /a0-a7)

Note that all the operands of the 68020 addressing modes are optional.

Data for the 68881 floating point instructions may be specified using floating point notation, i.e.

```
...#2.1  
...#2.1E+10
```

will be converted into the proper floating point formats according to the type of instruction. For example, in the following instruction:

```
fmove.s    #2.1,fp1
```

The 2.1 would be in single precision. Other sizes allowed are:

```
fmove.d    #2.1,fp1    ; double precision  
fmove.x    #2.1,fp1    ; extended precision
```

Note that the packed data format is not converted for you. Also if you want to specify the bit pattern by hand you may use the following formats:

```
fmove.s    #12345678,fp1    ; 32 bit  
fmove.d    #123456781234568,fp1    ; 64 bit  
fmove.x    #123456781234567812345678,fp1    ; 96 bit
```

You can also specify the constants in octal (i.e. @123456712) or binary (i.e. %0110110100110101).

Using the assembler from the command line

The assembler can be run via the following command:

```
asm [>listfile] [options] filename
```

Optional fields are enclosed in brackets, and all fields are described below:

>listfile

Causes the listing and error message output of the assembler to be directed to the specified file.

options

Assembler options are specified as a minus sign followed by a single letter; in some cases, additional text may be appended. The letter may be in either upper or lower case. Each option must be specified separately, with a separate minus and letter. The options are:

- d This option has two uses. It activates the debugging mode (in the same way as the compiler `-cl` option) or it defines symbols. When used to define symbols it may be used in the following ways.

-dsymbol

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

```
symbol EQU 1
```

-dsymbol=value

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

```
symbol EQU value
```


-ipfx Specifies that **INCLUDE** files are to be searched for by prefixing the filename with the string *pfx*, unless the filename in the **INCLUDE** statement is already prefixed by a drive or directory specifier. Up to 16 different *-i* strings may be specified in the same command. No intervening blanks are permitted in the string following the *-i*. Note that if a directory name is to be specified as a prefix, a trailing backslash *must* be supplied.

When an unprefixd **INCLUDE** filename is encountered, the current directory is searched first; then file names are constructed and searched for, using prefixes specified in *-i* options, in the same left-to-right order as they were supplied on the command line.

-lopt Causes a listing of the source file to be written to the standard output. The listing displays the appropriate program counter and code information alongside the assembly source. One or more of the following characters may be appended to the *-l* option, with the following effects:

i List the source for text from **INCLUDE** files as well as the original source file.

m List additional data generated for source lines which cannot be accommodated alongside the original source line (i.e. allows multiple listing lines for each source line).

x List the expansion text for macros.

-m This option controls whether warnings are generated when code for the relevant processor is encountered. The *-m* must be immediately followed by one of the letters from the following list:

0 Used for 68000 target. Provides warning if you attempt to use 68010/020/030/040/332 only instructions. This is the default case.

1 Used for 68010 target. Provides warning if you attempt to use 68000/020/030/040/332 only instructions.

2 Used for 68020 target. Provides warning if you attempt to use 68000/010/030/040/332 only instructions.

- 3 Used for 68030 target. Provides warning if you attempt to use 68000/010/020/040/332 only instructions.
 - 4 Used for 68010 target. Provides warning if you attempt to use 68000/010/020/030/332 only instructions.
 - 32 Used for 68332 target. Provides warning if you attempt to use 68000/010/020/030/040 only instructions.
 - 8 Used for 68881/68882 target.
 - 9 Used for 68851 target.
- nsig** Sets the significance of symbols to `sig` characters. If no size is specified, this option defaults to using 8 character significance.
- opfx** Specifies that the output filename (the `.O` file). If a directory name is specified the output name is formed by prefixing the input filename (the `.S` file which is being assembled) with `pfx`. Any drive or directory prefixes originally attached to the input filename are discarded before the new prefix is added. No intervening blanks are permitted in the string following the `-o`. Note that if a directory name is to be specified as a prefix, a trailing backslash *must* be supplied.
- u** This option automatically prefixes all external references with an underline (`_`). If references to C labels have already been prefixed with an underline, the option is not needed.
- w** This option works like the option `-dSHORTINT`.

filename

Specifies the name of the source file to be assembled. This is the only required field on the command line. If the name does not have an extension `.S` is assumed. The object file will have the same name as the source file, except that the source file extension is replaced with `.O`.

For example, the following command causes the assembly language source file `modn.s` to be assembled, producing the object file `modn.o`. A listing of the source file, along with any error messages generated, will be written to the file `modn.lst`.

```
asm >modn.lst -l modn
```

Assembler directives

The assembler handles all the instructions of all members of the M68000 family as detailed in the 'Motorola M68000 family programmers reference manual'. Assembler directives are instructions to the assembler rather than instructions to be translated directly into object code. Note that although the `IDNT`, `PAGE`, `SPC` and `TTL` directives are recognised, they are not supported and do not cause errors to be generated in order to provide compatibility with other assemblers. Also, as with instruction mnemonics, directives cannot begin in the first character of the source line.

COMM symbol,size

The `COMM` directive creates a 'common' block identified by `symbol` and of the given `size`. Space for a common block is allocated at link time and, in the absence of an external definition, is the size of the largest block encountered by the linker.

CNOP offset,alignment

This directive aligns the program counter using the given byte alignment and offset. For example,

```
cnop      1,4
```

aligns the program counter one byte past the next long-word boundary relative to the start of the current section. Note that

```
cnop      0,2
```

is equivalent to the `EVEN` directive found in other assemblers and will ensure that the following data is aligned on an even address (i.e. a word boundary). This is normally only necessary when 68000 instructions follow byte-aligned data as the `DC` and `DS` directives word-align automatically.

CSECT *name[,type,alignment,reltype,relsize]*

Defines a program control section. Some form of section *must* be defined before any data can be generated. All parameters are optional except *name* and have the following functions:

name	is the control section name, note that this is case sensitive.
type	may be CODE (or 0) for instructions, DATA (or 1) for initialised data, or BSS (or 2) for uninitialised data sections; the default value is 0.
align	specifies the alignment requirements of the control section as a power of 2; this parameter is currently ignored and all sections are longword aligned.
reltype	specifies the relocation type, which determines the default addressing mode to be used for all symbol references and definitions from within the control section. The default value is 0.
relsize	specifies the size, in bytes, of the relocation data for the section; the default value is 4.

Legal **reltype** and **relsize** combinations for relocation information on the 68000 are summarised in the following table:

reltype	relsize	Description
0	4	Absolute long addressing (default)
0	2	Absolute short addressing
1	2	PC-relative offset (PC)
2	2	Address-register-relative offset (A4)

A discussion of the use of CSECT directives which are compatible with the **-b** and **-r** options of the C compiler appears later.

[label] DC.B *expression[,expression] ...*
[label] DC.W *expression[,expression] ...*
[label] DC.L *expression[,expression] ...*

These directives define constants in memory. They may have one or more operands, separated by commas. The constants and any associated label will be aligned on a word boundary for DC.W and DC.L. You may also specify string expressions for DC.B within single or double quotes.

Be very careful about spaces in DC directives, as a space is the delimiter before a comment. For example, the line:

```
dc.b      1,2,3 ,4
```

will only generate 3 bytes - the ,4 will be taken as a comment.

```
[label] DS.B      expression  
[label] DS.W      expression  
[label] DS.L      expression
```

These directives reserve uninitialised memory locations. Any label specified is set to the start of the area, which will lie on a word boundary for the DS.W and DS.L directives. If used within a BSS section, the reserved space is simply added to the section size and no object code is generated.

For example, each of these lines will reserve 8 bytes of space in different ways:

```
ds.b 8  
ds.w 4  
ds.l 2
```

END

Signifies the end of program source.

ENDM

Terminates a macro definition. Must be used after a MACRO directive.

```
label EQU      expression
```

This directive permanently assigns the value and type of a given label to be equivalent to the expression. If there is an error or forward reference in the expression, the assignment will not be made.

```
IDNT string
```

Currently ignored, provided for compatibility only.

```
INCBIN filename
```

Includes a binary file, verbatim, in the output file. Suggested uses include graphics data and ASCII files. You may specify a drive specifier and directory for INCBIN, otherwise it will default to searching the current directory.

INCLUDE filename

This directive will take source code from a file on disk and assemble it exactly as though it were present in the text. The directive must be followed by a filename in normal GEMDOS format. If a drive specifier or directory is included, the entire filename must be surrounded by quotes, e.g.

```
include "b:\constants\header.s"
```

In the absence of a drive specifier, the filename is taken to be relative to the current directory and any include directories specified on the command line are also searched.

Include directives may be nested up to 16 levels and if any error occurs when trying to open the file or read it, assembly will be aborted with a fatal error.

LIST

Turns on the assembly listing. All subsequent lines will be listed until an END directive is reached, the end of the text is reached, or a NOLIST directive is encountered.

[label] MACRO

This starts a macro definition causing all following lines to be copied into a macro buffer until a matching MEXIT directive is encountered. The presence of a label determines whether Motorola-style macros are to be used. Refer to the macro definition section for a more detailed explanation.

MEXIT

This can be used as part of a MACRO definition to stop the current macro expansion prematurely, usually as a result of a conditional. EXITM is accepted as a synonym for MEXIT.

NARG

This is not a directive but a reserved symbol. Its value is the number of parameters passed to the current macro. Note that \# may be used as a synonym for NARG.

NOLIST

Switches the assembly listing off.

OFFSET [expression]

The OFFSET directive switches code generation to a special dummy section for the generation of absolute labels. The optional expression sets the value for the first label, otherwise zero is used. No bytes are written to the disk and the only directive allowed is DS. This can be used to generate labels which represent offsets into a data structure. For example,

```
        offset    10
next    ds.l      1
title   ds.b      32
```

will assign the value of 10 to the label next and 14 to title (i.e. 1 longword after next). To return to ordinary code generation, use the CSECT or SECTION directive.

OPSYN name,opcode

Can be used to create a synonym of any valid label name for any opcode, directive or macro. Some examples of synonym definition and usage are:

```
        opsyn     banana,move
        opsyn     is,equ
        opsyn     .dcb,dc.b

label   banana.l  d0,d1
        is        42
        .dcb     1,2,3,4
```

The last example shows how this feature can be used to create pseudo-directives which provide compatibility with other ST assemblers in a way that is not possible with standard macro definitions.

PAGE

Currently ignored, provided for compatibility only.

RORG expression

This directive changes the program counter to the specified number of bytes from the start of the current section. Note that the value specified *must be less than* the current PC.

SECTION name[,type]

Define a program section. There are no restrictions on name and the optional type may be one of the following (in upper or lower case):

CODE	code section (instructions)
DATA	data section (initialised data)
BSS	BSS section (uninitialised data)

The default type is CODE. Note that the SECTION directive is a subset of the CSECT directive which is explained in greater detail elsewhere.

label SET expression

This is similar to EQU, but the assignment is only temporary and can be changed with a subsequent SET directive. Forward references cannot be used in the expression.

TTL string

Currently ignored, provided for compatibility only.

XDEF symbol[,symbol...]

Defined symbols may be exported using XDEF; the symbol type (relocatable or absolute) will also be exported.

XREF symbol[,symbol...]

This defines labels to be imported from other programs or modules. If any of the labels specified are already defined an error will occur, although importing a label more than once is accepted. Note that the symbol will inherit the relocation type of the control section in which it appears.

Conditional assembly

Conditional assembly allows the programmer to write a comprehensive source program that can cover many conditions. At the start of the conditional block there must be one of the many IF directives and at the end of each block there must be a corresponding ENDC directive.

IF	<i>expression</i>
IFEQ	<i>expression</i>
IFNE	<i>expression</i>
IFGT	<i>expression</i>
IFGE	<i>expression</i>
IFLT	<i>expression</i>
IFLE	<i>expression</i>

These directives evaluate the expression, compare it with zero and then conditionally assemble depending on the result. The conditions correspond exactly to the 68000 condition codes with the exception of the IF directive, which is identical to IFNE.

IFD	<i>label</i>
IFND	<i>label</i>

These directives allow control depending on whether a label is defined or not. With IFD, assembly is switched on if the label is defined, whereas with IFND assembly is switched on if the label is *not* defined.

IFC	<i>'string 1','string2'</i>
IFNC	<i>'string 1','string2'</i>

Primarily for use within macros, these directives perform a case-sensitive comparison of two strings, both of which must be enclosed within quotes. IFC will only assemble the block if the strings match exactly, whereas IFNC does *not* assemble if the strings match.

ELSE

Toggles conditional assembly on or off. If the preceding conditional block was assembled, ELSE will cause assembly to stop until a matching ENDC is encountered, and vice-versa. ELSEIF is accepted as a synonym for the ELSE directive.

ENDC

This directive terminates the current level of conditional assembly. If there are more ENDCs than IFs, an error will be reported.

Macro definition

Asm supports two styles of macro definition. Motorola standard macros are defined via the following sequence:

```
name      MACRO
...
ENDM
```

The definition must begin with the macro name followed by the directive `MACRO`. This is followed by the lines that comprise the macro itself, terminated by the `ENDM` directive. The `MEXIT` directive may also be used within the macro to terminate the macro early. Using this method of definition, macro parameters are referenced by a backslash and a number, for example

```
move.w    \2, (a0)
```

which would substitute the second macro parameter for `\2`. Alternatively, you may wish to use the second form of macro definition which is more flexible although non-standard:

```
MACRO
name      [arglist]
...
ENDM
```

With this system the `MACRO` directive must appear first, followed by a line showing a model of how the macro will be called. The `arglist` is a comma-separated list of argument strings which provide macro parameter names and default values in the following format:

```
arg[=default]
```

where `arg` is an identifier which can be used within the macro to refer to the corresponding argument text in the macro invocation and `default` is a string that will be associated with `arg` when that argument is not provided by a particular macro invocation. Note that `default` must be enclosed in single or double quotes if it contains any white space characters.

Both formats of macro definition support the `NARG` reserved word - and its alternative syntax of `\#` - which will be substituted with the number of macro arguments. Also, quoted strings may be passed as macro parameters.

In order to define labels within a macro you should use the special symbol \@. This causes the assembler to generate a unique number each time the macro is used, preventing multiple definitions of the same label.

The following example illustrates macro definition using the second style:

```
MACRO
MINWORD      source=#100,dest
  cmp.w      source,dest
  blt.b      min\@
  move.w     source,dest
min\@
ENDM
```

The macro name is MINWORD and it could be invoked in the following way:

```
MINWORD      ,d2
rts
```

resulting in the instructions,

```
cmp.w      #100,d2
blt.b      min.0
move.w     #100,d2
min.0
rts
```

Note that the default value of #100 was substituted because the first parameter was omitted and that \@ was replaced by .0 (calling the macro a second time would use .1 etc.).

Pre-defined macros and synonyms

To aid in porting code from other assemblers there are a number of pre-defined macros and opcode synonyms which reflect common usage under other assemblers:

```
BSS      MACRO
          CSECT      udata,2
          ENDM
```

DATA	MACRO	
	CSECT	data,1
	ENDM	
EVEN	MACRO	
	CNOP	0,2
	ENDM	
TEXT	MACRO	
	CSECT	TEXT,0
	ENDM	
	OPSYN	ELSEIF,ELSE
	OPSYN	EXITM,MEXIT

Interfacing C with assembly language

The aim of this section is to discuss the conventions which a program must follow when interfacing to C. Attention is given to features of the Lattice assembler, *Asm*, which assist in writing such code and some of the pitfalls which can occur. Full examples of both C calling an assembly language routine and assembly calling a C function are given towards the end of the section.

The following list covers the main points which you should bear in mind when writing assembly code for use with C. Each of these is covered in greater detail with examples later in the section.

- Separate control sections containing definitions or external references should be defined for code, initialised data and uninitialised data (BSS) via the `CSECT` or `SECTION` directives.
- Code references (including function calls) may use PC-relative addressing or branch instructions if the function is within a 32K range, otherwise you should use absolute addressing (i.e. a `JSR` instruction).
- Data references for *near* data should use register A4 as a base pointer whereas *far* data must use absolute addressing.
- Near data must be defined in the named section `__MERGED`.
- Standard argument passing functions are prefixed by an underscore (`_`) and use values pushed onto the stack.
- Register passing functions have a prefix of `@` and place *some* arguments in registers with the remainder on the stack.

- The `__asm` specifier can be used to determine which register each function argument is passed in, with certain limitations.
- The size of type `int` may vary between word and long. Also, type `char` may be signed or unsigned depending upon compiler options.
- Return values appear in D0 with D1 also being used for double values. Note that the condition codes after a function call *cannot* be relied upon.
- A function may only corrupt registers D0-D1/A0-A1, all others *must* be preserved, including 68881 floating point registers (except for FP0/FP1) if used.

Control sections

In order for an assembly language program to link correctly with C object files you must use named control sections. The Lattice assembler provides this facility through the `SECTION` and `CSECT` directives. The latter of these provides more powerful options concerning automatic conversion of addressing modes, although in many cases you can simply use `SECTION`. A summary of both options can be found in the assembly directives section.

Programs should be divided into *code* (assembly language instructions and routines), *data* (initialised data and constants) and *BSS* (uninitialised data) sections. Each of these is described in greater detail below.

Code sections

All assembly language instructions should appear within code sections. The two simplest form of directives you can use to specify a code section are:

```
SECTION    name
CSECT     name
```

where `name` is the control section name. The compiler uses the default section name of `text` for all code generation although you may wish to use different names to identify program modules.

Any functions defined within a code section can be called from the same module with a branch or jump to subroutine instruction which you may wish to make PC-relative. However, in order to make a function visible to other modules when the program is linked you must define it as an external definition, for example,

```
XDEF      newtable
```

would make the function `newtable` callable from any other module. You should take into account that the C compiler automatically prefixes all external references with an underscore character `_`. The `XREF` directive may be used to access an external reference which is defined in another module.

The `CSECT` directive may also be used to specify additional information about the control section; its general format is:

```
CSECT      name,type,align,reltype,relsize
```

Only the `name` parameter must be present; it specifies the name of the control section. The `type` parameter describes the type of section; code, data or BSS (the values 0, 1 and 2 may also be used). The `align` parameter specifies the alignment requirements of the control section. The last two parameters, `reltype` and `relsize`, specify the type and size of relocation information associated with symbols declared within the control section.

For example, the section directives described previously are equivalent to:

```
CSECT      name,code,4,0,4
```

which is interpreted as a named code section, aligned on a longword boundary, defaulting to absolute longword addressing for symbols. The final two parameters can be used in code sections to automatically convert absolute long addressing to PC-relative for more compact code, as in

```
CSECT      text,0,,1,2
XREF      _function
JSR      _function
```

Note that we have used the number 0 rather than `code` and the alignment parameter has been omitted as all sections are longword aligned. The `JSR` instruction will actually be assembled as

```
JSR      _function(PC)
```

because we have specified a relocation type of PC-relative. To override this you may move the XREF out of the PC-relative section. It is also possible to use several code sections with different relocation types, the assembler will only use PC-relative addressing for symbols declared in the correct sections.

The advantage of using CSECT to provide PC-relative instructions is that changing a single CSECT directive gives you the ability to transform all external references. This provides you with an equivalent mechanism to that provided by the `-r` option on `lc`.

To call a C function from an assembly language module, you must always include an XREF declaration for the function. Before calling the function (via JSR or BSR), you must supply any expected arguments in the proper order either on the stack or in registers, depending upon the style of parameter passing employed by the function. After control returns from the called function, the stack pointer must be adjusted to account for any pushed arguments.

```
XREF      _cfunc

MOVE.L   D0,-(A7)      ;push argument
MOVE.L   D1,-(A7)
JSR      _cfunc        ;call function
ADDQ.W   #8,A7        ;restore stack pointer
```

This code fragment illustrates stack parameter passing, more details can be found in the relevant section. Remember to prefix function names with an underscore `_` or `@` symbol accordingly.

Data sections

There are two types of control sections in which program data can be held; *data* and *BSS* sections (described later). The first of these is for initialised data and constants and may be defined with either of the following directives,

```
SECTION   name,data
CSECT    name,data
```

where *name* is the control section name. The compiler uses two names for data sections; *data* for far data (this is accessed with absolute long addressing) and `__MERGED` (the program's near data, accessed as a base-relative offset from register A4). Examples of instructions used to access each type of data are

```
move.w    fardata,d0
move.w    neardata(a4),d1
```

When defining global data in assembly which is accessed by a C program you must declare the symbol as an external with an `XDEF` directive. The C source must also include an `extern` declaration of the correct type. For example, this assembly program *defines* a global variable:

```
        CSECT    asmdata,data
        XDEF    _entrynum

_entrynum DC.W    15
        END
```

Note that data is always prefixed with an underscore. This can be done automatically via the `-u` option. The corresponding C code to declare the variable is as follows,

```
extern unsigned short far entrynum;
```

The Lattice assembler provides a way of specifying a near data section, i.e. where all the data lies within a 32K range which is accessed off A4. All absolute longword references to symbols declared within such a control section will automatically be converted to the address-register-relative addressing mode. This is done through the `CSECT` directive:

```
CSECT    __MERGED,data,,2,2
```

where the case of the section name *is* important. In practice, this gives you a direct equivalent to the `-b` option of `lc`, allowing you to change the arrangement and thus the access mode for any data by simply placing it in an appropriate control section. Consider the following code:

```
SECTION    text
move.w    globl,d0
move.l    _otherdata,d1
rts

CSECT    __MERGED,1,,2,2
```



```

XREF      _otherdata
globl     DC.W      42

```

The move instructions will actually be assembled as

```

move.w    global(a4),d0
move.l    _otherdata(a4),d1

```

because the symbols were declared in a near data section.

BSS and offset sections

The second form of data section is the *BSS* or uninitialised data section. It behaves in exactly the same way as a regular data section except that the only directive allowed is the *DS* directive. By placing all data which you require to be initialised to zero in the *BSS* section you can save considerable file space because no data is actually written, the *size* of the section is merely remembered.

The directives to start a *BSS* section are identical to data sections in every respect other than the section type. The special section name of `__MERGED` is also recognised for near data in a similar way to that described previously.

Although visibly very similar to a *BSS* section, an *offset* section describes merely the layout of data and not actually a specific instance of it. The primary use of the *OFFSET* directive is to provide a simple way to declare offsets into data structures. For example, here is a structure described in C:

```

struct NameNode {
    struct NameNode *next;
    struct NameNode *prev;
    int uses;
    unsigned char name[16];
};

```

In order to use this structure from an assembly language program, we must use numerical offsets into the structure. To aid readability and maintainability we wish to use symbols which refer to each element. The following description provides just that:

```

OFFSET
nn_next   DS.L      1
nn_prev   DS.L      1
          IFD       SHORTINT
nn_uses   DS.W      1

```

```

        ELSE
nn_uses DS.L      1
        ENDC
nn_name DS.B      16
sizeof_nn DS.B    0

```

This does not generate any code, simply offset values. The symbols `nn_next`, `nn_prev` and `nn_uses` will be set to the absolute values of 0, 4 and 8 respectively. The prefix of `nn_` has been added to avoid possible name clashes with other symbols and the dummy entry `sizeof_nn` provides a convenient way of referring to the size of the entire structure.

A conditional block has been used around the integer field because the length of an integer may vary between word and longword. Using this method, re-assembling the source with the `-w` flag for short integers will automatically generate the correct offsets. Some code which accesses this structure might look like the following:

```

        lea        firstnode(a4),a0
        subq.w     #1,nn_uses(a0)
        move.l     nn_next(a0),a0
        rts

```

Function Entry Rules

There are several rules which the compiler enforces to provide a mechanism for calling functions. These rules must also be followed by assembly programmers wishing to interface with C.

Regardless of how the function was called, register A7 (the stack pointer) always points to a return address. Register A4 points into a program's near data to allow base-relative addressing as discussed in the previous section.

Depending upon the style of parameter passing employed by a particular function, parameters may either be found on the stack, in registers or a combination of both. Arguments are always passed by value. An explanation of the three methods of parameter passing follows.

Standard arguments

This is the default method of parameter passing where all function arguments are placed on the stack immediately before the return address. The `__stdcall` keyword may also be used in a function prototype or definition to force stack parameters. Note that functions which take a variable number of parameters *always* use standard argument passing.

Register A7 is the stack pointer which points to the 4-byte return address followed by the arguments in left-to-right order. Arguments can then be accessed as an offset from the stack pointer. The exact location of the parameters on the stack depends on the argument types and the current flags. Considering the default long integer mode, for the function call:

```
char ccc;  
double ddd;  
int iii;  
func(ccc,ddd,iii);
```

The compiler generates code to extend each of the parameters to the size of an `int` if it is smaller and then push the arguments onto the stack in *reverse* order. For example,

```
move.l    d0, -(sp)  
movem.l  d2-d3, -(sp)  
ext.w    d1  
ext.l    d1  
move.l    d1, -(sp)
```

This results in a stack organised in the following way:

Location	Size	Contents
(A7)	4	Return address
4(A7)	4	Argument ccc
8(A7)	8	Argument ddd
16(A7)	4	Argument iii

By comparison, in default short integer mode (option `-w`) the compiler would generate code to push the arguments `ccc`, `ddd`, and `iii` onto the stack using *two* bytes, eight bytes and *two* bytes, respectively:

```
move.w    d0, -(sp)  
movem.l  d2-d3, -(sp)
```

```

ext.w      d1
move.w    d1, -(sp)

```

Location	Size	Contents
(A7)	4	Return address
4(A7)	2	Argument ccc
6(A7)	8	Argument ddd
14(A7)	2	Argument iii

Note that due to the widening of `char` types to the size of an `int`, the actual parameter is in the *low byte* of the `int` although the full integer value may be used. Also remember that `char` may be signed (the default) or unsigned depending upon compiler options.

If a structure or union is passed by value to a function, then the contents of the aggregate are copied onto the stack with the last element pushed first. In effect you receive a complete copy of the aggregate on the stack followed by a single byte for alignment if necessary.

Stack space occupied by function arguments may be used by the function as temporary workspace once the values are no longer needed.

Register arguments

If a function is explicitly declared `__regargs` or is called from a module compiled with the `-rr` option, some arguments are passed in registers instead of on the stack. Note that functions which accept a variable number of parameters always use the previous style of parameter passing.

With register parameters, the first two pointer arguments will appear in A0 and A1, and the first two integral arguments will be in D0/D1 and widened to an `int` if necessary as previously described.

When not generating inline floating point code (`-f8`) a double pair will be passed in D0/D1 if both are available, or any combination of float and integral parameter may be passed in D0/D1. When inline FPU code *is* being generated then FP0/FP1 are used to pass real parameters.

Structures and unions along with any parameters not placed in registers are passed via the stack in the usual way.

Obviously, the function needs to know whether it is being called with some arguments in registers or with all arguments on the stack. The compiler helps make this distinction by placing the character @ in front of function names that are called with register arguments, replacing the underscore that the compiler normally supplies as a function prefix.

The `__asm` keyword

Providing much greater control over register passing, the `__asm` keyword allows you to specify exactly which registers parameters are to be passed in. It can be used in both function definitions and declarations:

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

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

In order for the register specifier sequence to be used, you must have the `__asm` keyword specified on the function. If you do use the `__asm` keyword, you *must* specify a register for each parameter and not re-use the same register for any two parameters. If you need to pass some parameters on the stack then you should use the `__regargs` keyword instead. Note that currently the compiler is restricted to returning only basic types like `long`, `double`, etc.

In order to permit the most flexibility in register passing, the compiler does not limit what registers may be passed. However this can lead to situations in which it is impossible to generate code that works in the presence of aliased variables. To ensure that such situations are not encountered, you should avoid utilising registers that would normally be assigned as register variables and instead only use the registers:

```
__d0      __a0      __fp0
__d1      __a1      __fp1
__d2      __a2      __fp2
```

The best advice is to be careful when using this feature and if you are uncomfortable with it, use the `-rr` option of `lc1` (or `__regargs`).

Another mechanism which may be used to achieve similar effect to `__asm` is the `#pragma inline` statement described in detail elsewhere in this manual. When no instruction stream is present, this will generate a function call which may use any register or the stack for parameters and may use any register for the return value.

Function exit rules

Function return values are passed back in one or more registers, depending on the data type declared for the function. The conventions are:

Return Data	Bits	Asm Syntax	Meaning
char	8	D0.B	Low byte of D0
short	16	D0.W	Low word of D0
long	32	D0.L	All of D0
float	32	D0.L	All of D0
double	64	D0.L, D1.L	High bits in D0
pointer	32	D0.L	All of D0

Note that the above table does not mention `int`. An assembly language function should return its value as a `short`, if in default short integer mode (`-w`) or as a `long` if not in that mode, i.e. `D0.W` or `D0.L`.

If inline floating point is being used (`-f8`) then register `FPO` is used for floating point returns, viz:

Return Data	Bits	Asm Syntax	Meaning
float	96	FPO	All of FPO
double	96	FPO	All of FPO

If the function returns a structure or union, it must define a static work area (i.e. not on the stack) to temporarily hold the returned object. Then the function must return in `D0` a pointer to this temporary copy, and the calling function will immediately move the data to the appropriate place. This approach implies that functions returning structures or unions are not re-entrant, although they are serially re-usable. Such functions *can* be recursive if designed very carefully with this in mind.

The registers D2 through D7 and A2 through A6 must be saved if they are used by the function, similarly if a 68881 maths co-processor is present (only possible on 68020 or 68030 systems) and any of the floating point registers FP2 through FP7 is used, they must also be saved.

After setting up the return value, a function exits with the RTS instruction. Note that the calling function removes the arguments from the stack.

Calling assembly from C

To illustrate how the rules governing C functions affect an assembly language routine we have chosen a short example which can be implemented either as C calling assembly, or assembly calling C (the C and assembly object modules must be linked with the startup code and appropriate libraries). It illustrates many of the points made previously and can be used as a basis for your own function calls.

The function returns a hash value calculated by adding together the ASCII codes of each character in the supplied string up to a specified length. This value is then divided by the number held in the global variable `maxhash` and the remainder (or modulo) is returned.

The calling program simply defines and initialises the variable `maxhash` and calls the hash function with a sample string. Implemented in C, this is as follows:

```
unsigned short maxhash; /* definition */
/* declaration (prototype) */
unsigned int
hash(unsigned int length, const char *string);

void
main(void)
{
    unsigned int result;

    maxhash = 101;
    result = hash(4, "Banana");
}
```

The hash function coded in assembly language for default addressing modes, parameter passing and types:

	CSECT	text,code	control section
	XDEF	_hash,@hash	declarations
	XREF	_maxhash	imported global
_hash			; stdargs entry point
	movem.l	4(sp),d0/a0	get the parameters
@hash			; regargs entry point
	move.l	d2,-(sp)	preserve register
	moveq	#0,d2	
	bra.s	1\$	
2\$	move.b	(a0)+,d1	
	ext.w	d1	
	ext.l	d1	
	add.l	d1,d2	
1\$	subq.l	#1,d0	
	bcc.s	2\$	
	divu	_maxhash(a4),d2	
	clr.w	d2	make result 32-bit
	swap	d2	get remainder
	move.l	d2,d0	return value in D0
	move.l	(sp)+,d2	restore register
	rts		
	END		

Any labels available to the C program are prefixed by an underscore character `_` or `@`. Note that for this function, it is easy to provide an entry point for register parameter calling by simply bypassing the code which loads arguments from the stack into registers for use by the body of the function. If you are using register parameters as default, you may leave out this code entirely.

The global variable is accessed as a base-relative offset from `A4` because we are using default near data. The function must also save `D2` on the stack because it is used as a temporary register and must be restored.

Compiling the program with default short integers, unsigned characters and far data does not change the C source although it causes many changes to the assembly language. The function must now be changed to:

```
_hash
    move.w    4(sp),d0    length is now a word
```



```

@hash      move.l      6(sp),a0      changing stack offsets
          move.l      d2,-(sp)
          moveq      #0,d1      can't sign extend char
          moveq      #0,d2
          bra.s      1$

2$        move.b      (a0)+,d1
          add.l      d1,d2
1$        dbra      d0,2$      optimised loop

          divu      _maxhash,d2
          swap      d2          don't clear high word

          move.w      d2,d0
          move.l      (sp)+,d2
          rts

```

Note that the parameters now have different offsets on the stack, characters can no longer be sign extended and global data must be accessed using absolute long addressing.

It becomes apparent that changes in compiler options such as `-b` or `-r` can dramatically alter the appearance of assembly code. The Lattice compiler provides some ways of insulating the programmer from these factors, as illustrated in the next section.

Calling C from assembly

This time, we will write the same program but as a C function called from assembly language. In order to provide the greatest flexibility whilst preserving code clarity, we will make use of the `CSECT` directive. This is the calling program for register arguments only:

```

          CSECT      text,code,,1,2
          XDEF      @main      PC-relative
          XREF      @hash

@main                                ; regards version
          move.w      #101,maxhash
          moveq      #4,d0
          lea      string,a0
          jsr      @hash      returns D0
          rts

          CSECT      __MERGED,data,,2,2

```

```

string    DC.B      'Radish'      data access off A4

          CSECT    __MERGED,bss,,2,2
          XDEF     maxhash
maxhash   DS.W      1
          END

```

Firstly, you may notice that there are no longer underscores before external labels. This is because the assembler can be called with the `-u` option which automatically prefixes an underscore to all externally visible labels whilst being overridden by the presence of an `@` symbol.

The relocation type and size parameters of the `CSECT` directive have been used in order to provide automatic PC and A4 base-relative addressing modes for the relevant sections. This has the effect of automatically converting the references to `string` and `hash` to:

```

      lea    string(a4),a0
      jsr    hash(pc)

```

Simply changing the relocation type allows the assembler to automatically generate the correct addressing modes. This corresponds directly to the C compiler options. To override the default addressing mode you may simply specify another, or for external symbols, provide an `XREF` in an appropriate control section. In our example, moving the reference to `@hash` outside the PC-relative section forces absolute long addressing for all references to that symbol.

Specifying the special section name of `__MERGED` causes the linker to include the section contents within the program's near data segment allowing base-relative addressing via A4.

Now the hash function written in C:

```
/* declaration */
extern unsigned short maxhash;

unsigned int __regargs
hash(unsigned int length, const char *string)
{
    unsigned long total = 0;

    while (length-- > 0)
        total += *string++;

    return total % maxhash;
}
```

The `__regargs` keyword is present to force the compiler to use register passing for this function. Remember to link with the startup code and libraries for register parameters since we are using `@main` rather than `_main`.

Asm error messages

1 invalid opcode

An unrecognised opcode name was encountered; this is often caused by a mis-typed instruction.

2 unrecognized opcode

An operation was encountered which was not recognised as a valid opcode, synonym or macro.

3 data generation must occur in reloc section

A data generation operation other than DS appeared in an OFFSET section.

4 invalid operands for this opcode

This error can be caused by invalid addressing modes, data size, macro parameters etc.

5 (W)

label ignored <label>

The label before a directive, such as a conditional, is not a recognised syntax and has been ignored.

6

must occur inside section

A data generation directive was used outside a control section.

7

invalid symbol

A symbol containing an illegal character or characters was declared.

8

public symbol not defined <name>

The program source contained an XDEF directive of a symbol which was not defined in the program.

9

cannot define absolute public symbol

10

external symbol redefined <name>

11

invalid expression

An OFFSET or IF directive contained an invalid expression. This error can also be caused by an expression containing a divide or modulo by zero.

12

missing label

An EQU or SET directive was encountered with no corresponding label.

13

duplicate label <label>

More than one definition of the same label was encountered.

14

not inside scope of IF directive

An ELSE directive was found which did not lie within a conditional control block.

15

invalid origin

An assembly directive causing incorrect data alignment or origin was found.

16 **constant size not same as relocation size**

A reference to a relative symbol conflicts with the byte size of relocation information for the current control section.

17 **invalid string**

A define constant or condition directive contained an invalid string.

18 (W) **extraneous data on input line**

A valid source line was followed by invalid text, which was ignored. This can be caused by providing too many parameters for an assembler directive.

19 **duplicate section name**

A section name was re-used illegally. This error can also be caused by an invalid OPSYN directive.

20 **ELSE/ENDIF not found**

An unterminated IFCC directive was encountered.

21 **label offset different in pass 2**

A phasing error caused by different code being generated on the first and second pass.

22 **macro argument too large**

A macro invocation was encountered with an argument string which was too long.

23 **missing macro definition**

The definition of a macro could not be found.

24 **illegal macro definition**

The syntax of a macro definition was incorrect.

25 **duplicate macro definition**

A macro was defined more than once.

26 **invalid control section parameter**

A CSECT directive with invalid parameters was encountered.

- 27** **invalid file name**
The filename specified for an INCBIN or INCLUDE directive was not valid.
- 28** **maximum include file nesting exceeded**
The INCLUDE directive has nested files too deeply. This is caused by included files referencing other files to a number of levels.
- 29** **file not found**
The file specified by an INCLUDE directive could not be found.
- 30** **invalid repeat count**
- 31** **macro substitution line overflow**
The substitution of macro arguments caused the line to overflow.
- 32** **immediate data out of range**
An arithmetic or logical operation was specified with an out of range immediate value.
- 33** **invalid effective address for opcode**
An attempt is being made to use an addressing mode not supported by the current instruction.
- 34** **invalid instruction size**
An attempt is being made to use an instruction size not supported by the current instruction.
- 35** **target out of range**
A reference to a label which is out of range has been encountered.
- 36** **value out of range for addressing mode**
An out of range value was used in an addressing mode.
- 37** **input line buffer overflow**
A source line has exceeded the maximum length.

38 **long branch to XDEF not supported by linker**

The Lattice linkable object file format does not support branches to external labels using a long-word offset.

39 **macro buffer overflow**

A macro definition was too long.

40 **ENDM/MEXIT not inside macro definition**

An ENDM or MEXIT assembly directive was encountered outside a macro definition.

41 **target not in current section**

42 (W) **END directive assumed**

This warning notifies you that there was no explicit END directive in the source file being assembled.

43 **invalid relocation type/size combination**

The specified relocation type and relocation data size specified in a CSECT directive are not available.

44 **unknown segment type**

The type specifier for a SECTION or CSECT directive was other than CODE, DATA or BSS.

45 **numeric value out of range**

A value in an expression overflowed the allowable range.

46 **opcode generated for <processor>**

An opcode only permitted for the indicated processors was generated.

47 **unrecognized expression**

48 **syntax error**

49 **invalid operation for relocatable data**

50 **undefined symbol**

51 **invalid opcode parameter reference**

52 **location counter not defined**

DERCS

The Resource Decompiler

Introduction

DERCS is utility for turning a resource file created using WERCS into a set of initialised data structures (OBJECT, BITBLK, ICONBLK etc.) which may subsequently be compiled to give a resource file embedded in a program.

This is advantageous when creating desk accessories (since desk accessories should not call `rsrc_load()`) and *essential* for writing control panel extensions (CPXs).

DERCS supports generation of both C and assembly language. Unfortunately the initialisation support available from other languages is not sufficiently rich to allow the representation of general resource files and so if a language of anything other than C is selected DERCS will generate an assembly language file.

Running DERCS

DERCS is run using a command line of the form:

```
dercs [-options] filename [filename]
```

The options are denoted by a - sign then a character *before* the filename. The options recognised are:

- a don't generate ANSI style function definitions. Normally DERCS generates any functions which it requires using the ANSI prototype syntax, viz:

```
void rsrc_init(void);
```

specifying this option causes it to use the old K&R syntax for function definitions. Obviously this flag has no effect for assembly language.

- cx specify casing is to be performed according to *x* rather than the value set in the .HRD file; the values used for *x* are identical to those used by WERCS:

0	mixed
1	upper
2	lower

- dx specify *x* as name of the section data is to be placed in. This option allows you to configure the name of the section which DERCS generates when outputting the data section information for assembly language. If none is specified it defaults to DATA. Note that Lattice C ASM users may like to change this to '___MERGED, data'. Obviously this flag has no effect for C.

- f Suppress output of tree fixup code. Normally DERCS generates a function which you may call which 'fixes up' the resource trees in your file (i.e. the conversion from character to pixel co-ordinates). Specifying this options suppresses this behaviour. You can use this option if you want to fix up trees yourself, or if you are using more than one resource file.

- h write only a header file. This option writes only a header file in the desired language. Note that with careful use of the -l option this allows the generation of resource file constants for more than one language (e.g. in a mixed assembler/C project).

- lx use language specified by language number *x*; the values used for *x* are identical to those used by WERCS:

1	C
2	Pascal
4	Modula-2
8	FORTRAN
16	Assembler
32	BASIC

Note that for DERCS values 2 through 32 *all* generate assembly language.

- px* use *x* as the prefix for automatically generated names. When DERCS is generating some of the more complex AES object structures it has to generate its own names for items which are unnamed in the resource file. If you are merging two or more DERCS'd resource files then this option can be used to ensure that no naming clashes occur.
- v* suppress output of `section` directives or `include` directives. This means that DERCS will not output any of the 'padding' which it normally generates.
- xic* generate pre-fixed resource file. This option is designed for building CPXs and fixes up all characters based on an 8x16 character. If *ic* is supplied, this should be the name of a free image in the resource file which is to be extracted into a .ICN file ready to be passed to CPREFIX.

The file specified first on the command line is then decompiled into either a C source file (if C was selected as the language in WERCS), or into assembler otherwise.

An optional second file name may be supplied to indicate a name for the output file.

Programming with DERCS

Programming using a resource file passed through DERCS is not dissimilar to using an ordinary resource file, the main difference is that `rsrc_gaddr()` is never used. Instead, DERCS builds all of the necessary resource information into a C file with the extension .C. This file also contains a special resource intialisation routine `rsrc_init()` (this routine actually performs the normal character to pixel coordinate transformations made by `rsrc_load()`).

Consider the following fragment written without DERCS:

```
#include <aes.h>
#include "resource.h"

int main(void)
{
    OBJECT *myobj;

    appl_init();
    rsrc_load("resource.rsc");
}
```

```

    rsrc_gaddr(TREE, MYOBJ, &myobj);
    objc_draw(myobj, ROOT, ...);
    ...
    rsrc_free();
    appl_exit();
}

```

Now assume that the command:

```
dercs resource.rsc
```

had been issued, generating the files `resource.c` and `resource.h`, then the same program could be written with DERCS:

```

#include <aes.h>
#include "resource.h"
#include "resource.c"

int main(void)
{
    appl_init();
    rsrc_init();
    objc_draw(MYOBJ, ROOT, ...);
    ...
    appl_exit();
}

```

Note that the `rsrc_load()` call has been replaced by `rsrc_init()`, and that the `rsrc_gaddr()` and `rsrc_free()` have disappeared altogether!

The object `MYOBJ` (which was what you named your object in `resource.rsc`) which is normally declared as a `#define` constant by `WERCS` is replaced by an `OBJECT` definition by `DERCS`, hence the address of the object (which `rsrc_gaddr()` would have obtained) is found simply by naming the object.

CPXBUILD

CPX .HDR Utility

Introduction

CPXBUILD is used to build the .HDR files used as arguments to the linker PREFIX option which specify the parameters for a CPX.

Running CPXBUILD

CPXBUILD is run using a command line of the form:

```
cpxbuild [-options] filename
```

The options are denoted by a - sign then a character *before* the filename. The options recognised are:

- b set boot-init flag; the boot initialisation flag of the CPX is set indicating that the CPX should be called during XControl's initialisation.
- col* set title colour; the colour of the title is set to *col*. The colour numbers used are those used by the AES.
- did* set CPX-id; the CPX id is set to *id*. If *id* has the special format '*cpid*' then the specified alphanumeric *cpid* is used as the long-word CPX id.
- fval* set flags absolutely; the header flags are set to the absolute value *val*; this option is provided to support any future additions which Atari may make to XControl during the lifetime of a CPXBUILD release.
- icn* specify an icon file; *icn* specifies a .ICN file which is to be used as the CPX's icon. The .ICN file is generated using DERCS' -*xic* option.

- ntxt* set icon text; the descriptive text attached to the icon is set to *txt*.
- pcol* set icon colour; set icon colour; the colour of the title is set to *col*. The colour numbers used are those used by the AES.
- r* set resident flag; the resident flag of the CPX is set indicating that the CPX should be made resident at the time of XControl's startup.
- s* set set-only flag; the set-only flag for the CPX is set; this indicates that the CPX performs all its work at boot time and need never be called again.
- txt* set title text; the title text for the CPX is set to *txt*.
- V* force CPXBUILD to print sign-on and version numbers. Normally CPXBUILD runs silently; this option causes more information to be generated.
- vvsn* set CPX version; the CPX version number is set to *vsn*. Typically this will have the format *major.minor*.

The *filename* given on the CPXBUILD command line indicates the name of the .HDR file which should be built. Note that the .HDR is *not* automatically supplied; you must specify it if required.

Appendix A

Project file syntax

A project file is the file used by the integrated compiler to control the management of multi-module programs. These project (.PRJ) files are in an ASCII format and are compatible with those used by the German Pure-C™ and PKS-Edit™. The syntax may be described by the following grammar:

project:

*(* | filename) (options) = { module [(dependents)] }*

options:

.L [linker-options]
.C [compiler-options]
.S [assembler-options]
.A [librarian-options]

module:

c-source-file [compiler-options]
assembler-source-file [assembler-options]
linker-file | with-file | project-file

c-source-file:

** | filename | filename.C*

assembler-source-file:

filename.S

linker-file:

filename.O | filename.LIB

with-file:

filename.LNK

project-file:

filename.PRJ

dependents:

filename [, dependents]