Volume I

User Manual

Lattice C

The C system for your Atari ST

Volume I
User Manual

Published by HiSoft

Version 5
First edition March 1990 (ISBN 0 948517 29 8)
Second edition April 1991

ISBN for this volume 0 948517 37 9

ISBN for complete 3 volume set 0 948517 28 X

Set using an Apple Macintosh™ and Laserwriter™ with Microsoft Word™ and SuperPaint™.

Table of Contents

Introduction

Welcome to Lattice C Version 5, one of the most powerful and flexible programming environments available for the Atari ST/TT range of computers. This new version of Lattice C is based on proven Amiga and PC compilers, coupled with a complete suite of programming tools and libraries, resulting in a C development system that is unparalleled in the speed and the quality of its object code, while being a joy to use for the beginner and the professional alike.

The documentation for Lattice C 5 is divided, like Gaul, into three parts: this volume gives an overview of the whole system and details the working of the compiler, linker, assembler and all the associated tools while the other two volumes document the many library functions available, both generic (Unix, ANSI and Lattice) and Atari ST specific (GEM, AES, VDI etc.). There is a wealth of information here and you should not expect to assimilate it all in one reading - you are encouraged to work through the rest of this chapter and then to use Lattice C 5 in anger, dipping into the various manuals for reference, as and when the occasion arises.

Before we proceed to using Lattice C 5 for the first time there are a couple of important things that you must know about ...

a

You have several ways of making a backup of the Lattice C 5 disks. You can use the disk copying function of the Desktop to duplicate the disk, or you can use one of the many disk copiers available.

However you do it, *please make a backup* and then store the master disks in a safe place, away from moisture, extreme heat or cold, magnetic fields (televisions, telephones etc. give off radiation harmful to disks), strong light, coffee and, above all, children and dogs! If you damage your master disks we will charge you a handling fee for re-copying them.

We are striving continually to make this product better, so we are very receptive to suggestions about how Lattice C 5 could be made more useful to you. If enough people ask for the same types of features, the likelihood is high that such features will be implemented in a future release of the package.

To take advantage of the support we offer, even if only to receive details of the newest major revision of the program, you must have sent us your registration card. You will then be sent any new information about Lattice C.

You must also quote your serial number for technical support, you may find it useful to make a note of it here:

Serial No.

See **Appendix J** for more details of Technical Support. Please ensure that you have read this section *carefully* before contacting us for technical support as it also describes some of the common problems and how to solve them.

What is Lattice C 5?

So that you can get the most out of these manuals and the software, here is an overview of the complete Lattice C 5 system.

Lattice C 5 is a C development system comprising a host of tools and utilities allowing you to create, compile, link and run programs on the Atari ST. It conforms closely to the new ANSI standard for the C language and includes many extensions that give you a great deal of flexibility in the C environment. The package also contains the most extensive set of library functions (both generic and ST-specific) available on the Atari ST computer.

Editing C Source Code

You can create and edit C programs using any editor of your choice, as long as the editor can save the source code as plain ASCII characters. We provide, and recommend, an editor, called EdC, which is GEM-based, easy-to-use and is a complete, visually-oriented shell.

We also supply a derivative of EdC, called LC.PRG, which integrates the editor and the first phase of the Lattice C 5 compiler, giving a fast edit-compile-edit cycle. This is usable on machines with at least 1Mb of memory.

Compiling and Linking C Source Code

In order to turn your C source code into an executable program, you need to compile and link it. If you are not using the integrated system (LC.PRG), you will need to use the compiler driver (LC.TTP) either from the Desktop or from a command line shell (CLI) - we supply a MS-DOS style shell which is known as Batcher.

The Lattice C 5 compiler operates in two phases, the first phase (LC1) performs pre-processing and parsing of your C code into an intermediate format known as a quad (.Q) file. The second phase (LC2) takes this quad file and produces 680x0 object code in a file (.O extension), ready for linking.

The linker, CLink, creates a runnable, machine code program by combining one or more object files produced by LC2 with the relevant supplied libraries.

The integrated system lets you, with a single key-press, invoke the first phase of the compiler (which is built in), LC2 and then the linker, producing an executable program, ready to run from the editor.

From a shell, the LC driver (LC.TTP) provides one command line to control and invoke LC1, LC2 and the linker.

Debugging the Program

We supply a low-level debugger (MONST2C.PRG) that will help the more experienced of you to investigate the operation (or not!) of your program. The debugger has some higher-level features, normally only found in source level debuggers, allowing access to your source code on a line number basis. We hope to produce a full source level debugger at a later stage.

Improving the Program

We supply a number of powerful tools allowing you to create good-looking, fast and compact programs with a minimum of effort.

WERCS is a resource editor giving full access to GEM, making it easy to incorporate menus, dialog boxes and icons. It is compatible with earlier resource editors and also lets you include graphics from art packages.

The assembler, ASM, is a full 680x0 macro assembler, tailored to the Lattice C 5 environment, letting you exploit the 680x0 family of processors to the full.

To achieve maximum performance without resorting to assembly language, Lattice C 5 incorporates a global optimiser, GO. GO gives you the option of increasing the performance of your program with no programming effort, although it can take some time to complete its task!

Bells and Whistles

Lattice C 5 also comes with a plethora of minor tools, whose usefulness will depend on your own needs and experience. These include such things as a reset-proof RAM disk, a librarian, an object module disassembler etc. which are all fully documented in **The Lattice C 5 Tools** chapter.

A Lattice C 5 Tutorial

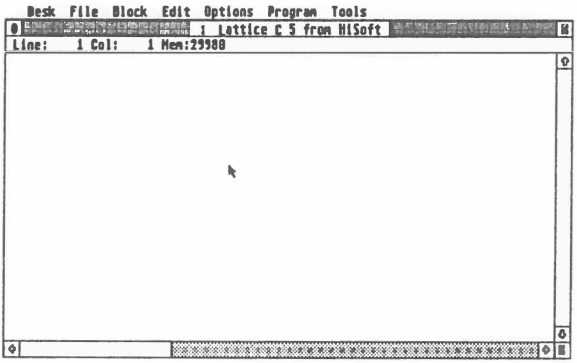
Lesson 1 - Your First Program

We are now going to guide you through your first experience with Lattice C 5 (at least, we hope it's your first experience - you haven't been too impatient, have you?). First of all, you must install Lattice C 5 for your system. The method of doing this depends on your system (how much memory you have and whether you have a hard disk or not) - we have documented the installation process in a separate document (the **Installation Guide to Lattice C 5**) since there are great number of different possible configurations and the installation may change as we upgrade the package.

Lattice C 5 will run on 512K machines but you would be well advised (as you will soon see) to upgrade your machine to at least 1Mb of memory so that you can take full advantage of the package's features. We will take you through a simple program on both 512K and larger computers to show you the different working environments.

520ST/STE Owners

First of all, find the EDC.PRG program (wherever you have put it on installation) and, from the GEM desktop, double-click on the EDC.PRG icon to run the program. The following screen will appear:

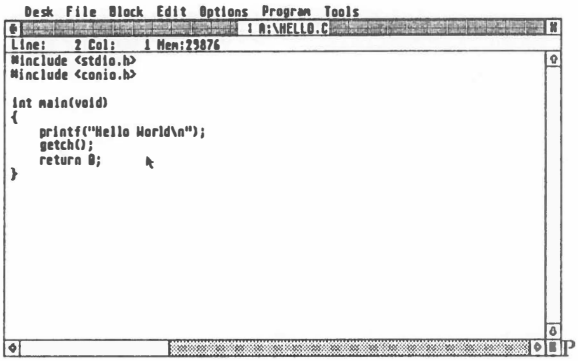


The EDC Editor Opening Screen

Now type the following program using the keyboard in the normal way and pressing Return at the end of each line:

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

int main(void)
{
  printf("Hello World\n");
  getch();
  return 0;
}
```

Your screen should now look like this:



The Hello World Program

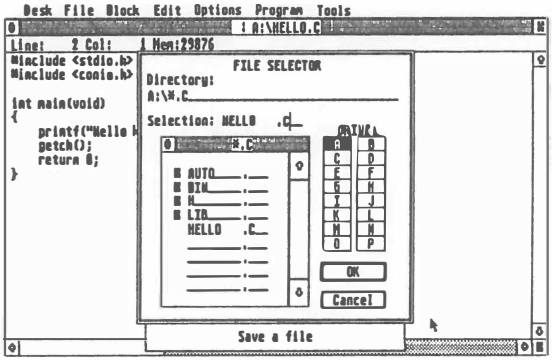
Before we compile this complicated program we must save it to disk because, on a 512K machine, we have to quit the editor to compile/link etc.

The best place to save your source files is the Lattice C 5 working disk or your hard disk if you have one. So press Alt-S or click on Save As... from the File menu and the file selector will appear.

The appearance of the file selector will depend on which one you have installed in your system. If your operating system is TOS 1.4 or greater, you have, automatically, an extended file selector with drive buttons such as that shown in the screen shot on the next page.

However, if you have an earlier version of the system ROMs, you will see, unless you have installed a new one, a much more primitive file selector, without drive buttons. You can replace this system file selector, if you wish, with the extended HiSoft File Selector (HFSEL) by placing HFSEL in your AUTO folder - see the **Installation Guide to Lattice C 5** for details.

When the file selector appears, type in HELLO.C and click on the relevant drive button so that the box looks like this:

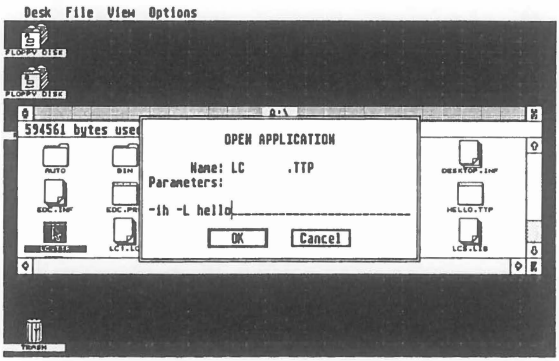


The File Selector

Quit the editor using Alt-Q or by selecting Quit from the File menu - you should now be back on the GEM desktop. Double-click on LC.TTP to run the compiler driver - a box will appear, type in:

```
-ih -L hello
```

so that the box looks like:



The LC.TTP Command Line for HELLO.C

The -Ih means 'find the include files in a directory called h' and the -L means 'link after compilation'.

Now press Return to start the compilation and link. Assuming this proceeds correctly you should see the following messages on the screen:

```
Lattice Atari C Compiler Copyright © 1990 HiSoft & Lattice, Inc.
All rights reserved - Version 5.04.00

Compiling hello.c
Module size P=0000001A D=0000000E U=00000000

Total files: 1, Compiled OK: 1
Linking hello
Clink Copyright © 1990 HiSoft & Lattice, Inc.
All Rights Reserved - Version 1.11

CLINK Complete - Maximum code size = 6464 ($00001940) bytes
```

The Compilation and Link of HELLO.C

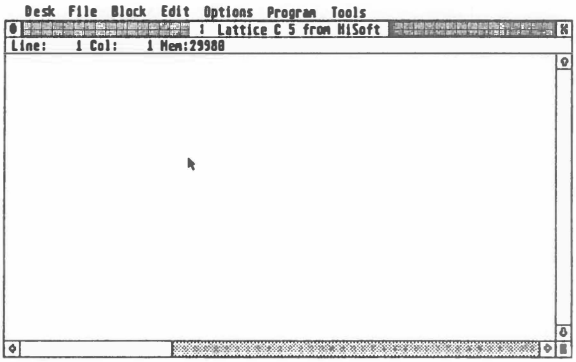
If you do not achieve a successful compile & link, check that you have installed the system correctly and that all the various tools and libraries are where we recommend.

After this you will have a file called HELLO.TTP on your working disk - double-click on this, type Return when the box appears and you should see Hello World appear at the top of the screen, hit a key and the GEM desktop will re-appear.

Congratulations - you have created, compiled, linked and executed your first Lattice C 5 program!

1040ST/STE, Mega, TT Owners

First of all, find the LC.PRG program (wherever you have put it on installation) and, from the GEM desktop, double-click on the LC.PRG icon to run the program. The following screen will appear:

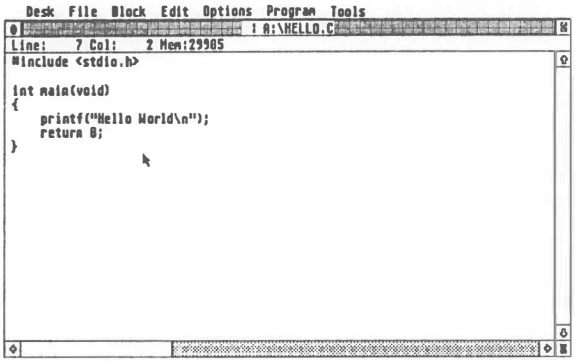


The LC Editor Opening Screen

Now type the following program using the keyboard in the normal way and pressing Return at the end of each line:

```
#include <stdio.h>

int main(void)
{
  printf("Hello World\n");
  return 0;
}
```

Your screen should now look like this:



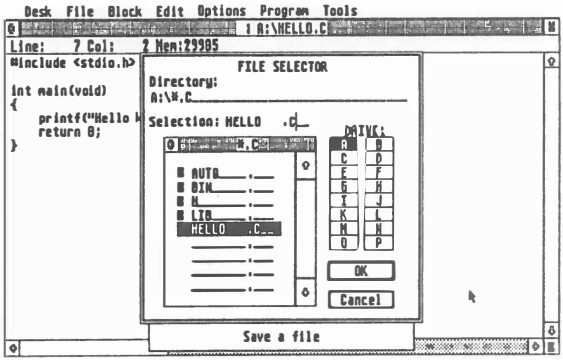
The Hello World Program

Before we compile this complicated program we really ought to save it to disk, in case anything goes wrong The best place to save your source files is the Lattice C 5 working disk or your hard disk if you have one. So press Alt-S or click on Save As... from the File menu and the file selector will appear.

The appearance of the file selector will depend on which one you have installed in your system. If your operating system is TOS 1.4 or greater, you have, automatically, an extended file selector with drive buttons such as that shown in the screen shot on the next page.

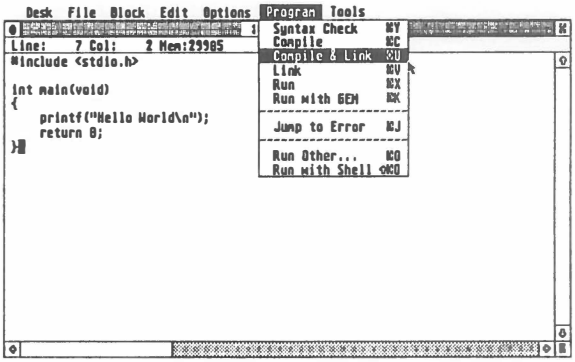
However, if you have an earlier version of the system ROMs, you will see, unless you have installed a new one, a much more primitive file selector, without drive buttons. You can replace this system file selector, if you wish, with the extended HiSoft File Selector (HFSEL) by placing HFSEL in your AUTO folder - see the **Installation Guide to Lattice C 5** for details.

When the file selector appears, type in HELLO.C and click on the relevant drive button so that the box looks like this:



The File Selector

To compile and link your program type Alt-U or select Compile & Link from the Program menu as shown below:



The Program menu before Compile & Link

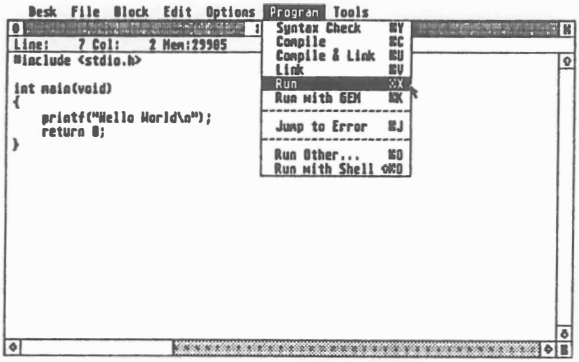
The compilation and link will now proceed immediately and, assuming it is successful, you will then be asked to press a key to return to the editor. If anything goes wrong, check that you have typed in the program correctly and that you have copied the EDCTOOLS.INF file as described in the **Installation Guide to Lattice C 5** - otherwise the environment may not be set up.

A correct compilation and link will generate messages something like this:

```
Lattice Atari C Compiler Copyright @ 1990 MiSoft & Lattice, Inc.
All rights reserved - Version 5.04.00

Module size P=0000001A D=0000000E U=00000000

Total files: 1, Compiled OK: 1
Linking hello
Clink Copyright @ 1990 MiSoft & Lattice, Inc.
All Rights Reserved - Version 1.11


CLINK Complete - Maximum code size = 6464 ($00001940) bytes

Final output file size = 6446 ($0000192e) bytes
```

The Compilation and Link of HELLO.C

After this you will have a file called HELLO.TTP on your working disk - to run the program first ensure that Run with GEM on the Program menu is *not* selected and then type Alt-X or click on Run on the Program menu, then type Return when the box appears and you should see Hello World appear at the top of the screen, hit a key and you will be returned to the editor.



Running your First Program

Congratulations - you have created, compiled, linked and executed your first Lattice C 5 program!

Summary of Lesson 1

So, what have we learned so far?

- 520ST/STE users cannot use a fully integrated system because there is not enough memory to support this - you must create/edit your program using EDC.PRG, exit and compile & link using LC.TTP, typing in the necessary command line.

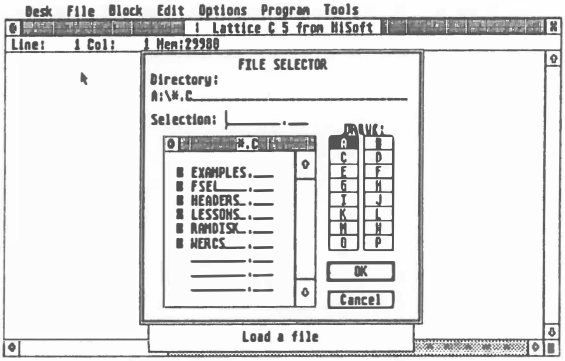
- *Important note for TOS 1.0 users:* this early version of the ST ROMs upper cases the command line typed in to .TTP programs. This can cause problems since the LC.TTP command line is *case sensitive* (because there are so many options). To get around this we have added an extra option, -?, that prompts for a further command line to be input, thus bypassing the ROM command line and allowing mixed upper and lower case to be used. Use -? if you have TOS 1.0 and need to use command line options that are case sensitive.

- 1040ST/STE and Mega users can take advantage of the integrated LC.PRG, creating, editing, compiling, linking and running their programs all from within the editor.

Lesson 2 - We all make mistakes

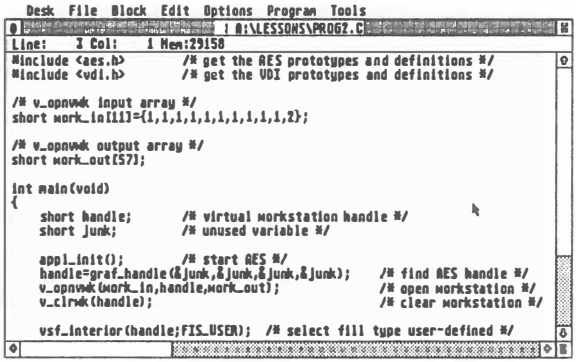
In this lesson we are not going to differentiate between 512K and 1Mb+ users and we shall simply tell you to compile, link and run your program - please refer to Lesson 1 if you are unsure how to do this.

Run EDC.PRG (512K users) or LC.PRG (1Mb+ users), insert your working disk and type Alt-L or select Load from the File menu



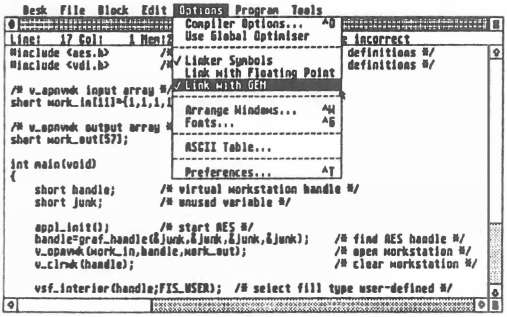
The Load File Command

Single-click on the Lessons directory to open it and then double-click on PROG2.C to load the program into the editor.

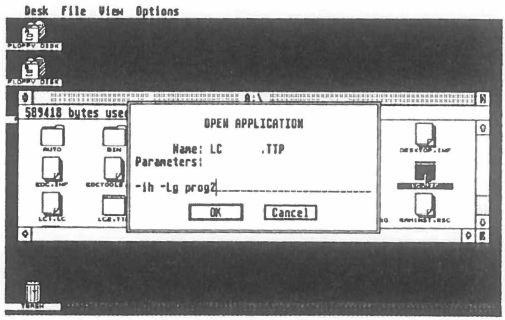


The PROG2.C Program

Now compile this program as you did in Lesson 1 but making sure that you select Link with GEM on the Options menu (1Mb+ users) or, if you are a 512K owner, use the -Lg option on the compiler command line (instead of just -L). (TOS 1.0 users will have to use the option -? and then type in the command line as described above).



Interactively Compiling with GEM



512K Users - Compiling with GEM

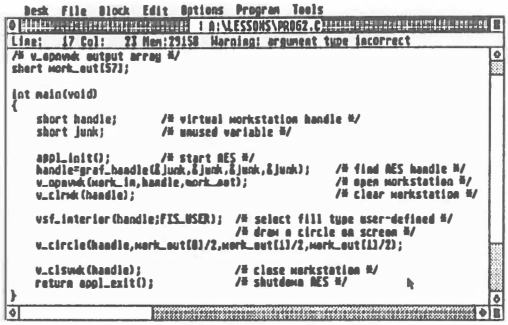
What happened? You should have seen a report something like this:

```
Lattice Atari C Compiler Copyright © 1990 HiSoft & Lattice, Inc.
All rights reserved - Version 5.04.08

        v_opnvwk(work_in,handle,work_out);                              /* open
workstation */
                          ^
A:\LESSONS\PROG2.c 17 Warning 88: argument type incorrect
        vsf_interior(handle;FIS_USER);  /* select fill type user-defined */
                           ^
A:\LESSONS\PROG2.c 20 Error 16: invalid function argument
        vsf_interior(handle; 4 );       /* select fill type user-defined */
                             ^
A:\LESSONS\PROG2.c 20 Error 57: semi-colon expected

Press any key
```

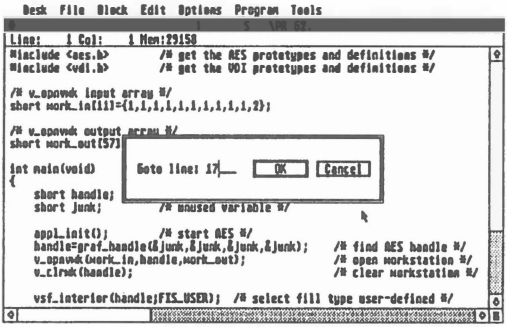
The first compilation of PROG2.C

The compiler has generated 1 warning and 2 errors.

Now get back to the editor with the PROG2.C program loaded. To achieve this, 1Mb+ users simply hit a key whilst 512K owners must run EDC.PRG and load PROG2.C. Now, how do we correct these errors? Well, again 1Mb+ users have all the luck - the cursor will already be positioned on the line in which the first error/warning occurred and the description of the problem will be in the top of the window. 512K users should note down the line numbers of the errors/warnings and then use the Goto... command on the Edit menu (or press Alt-G) to position the cursor in the first rogue line.



Interactively Getting to the Error



Getting to the Error for 512K Users

Looking at the first erroneous line (v_opnvwk(work_In, handle, work_out);) it looks legal, but the compiler has given an argument type Incorrect warning. This means that the type of one of the arguments in a function call was not what the compiler expected it to be. In fact, the problem here is that the second parameter in the v_opnvwk function has been declared in the header file as a pointer to a short and, as such, the address of handle (&handle) should be passed.

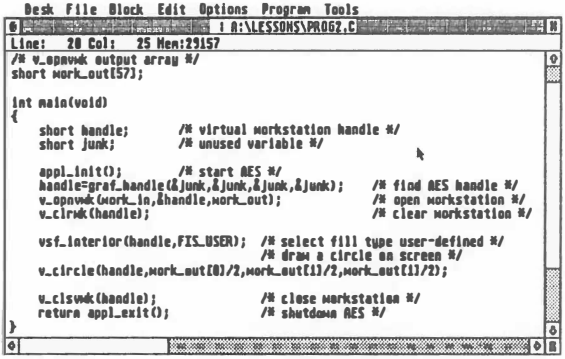
So position the cursor on the h of handle and type an ampersand character, &.

Now go to the next error by pressing Alt-J or Alt-G (depending on your memory - in more ways than one!), this is in the line:

```
vsf_interior(handle;FIS_USER); /* select fill-type */
```

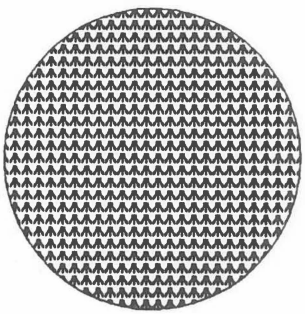
and the error is Invalid function argument. Well, it shouldn't take you too long to spot that the semi-colon between handle and FIS_USER should be a comma. Position the cursor over the F of FIS_USER and hit Backspace followed by a comma - ,.

Now compile the program again, remembering to save it first. It should now compile and link successfully, to PROG2.PRG, and you can try running it.



The Corrected Program

The program draws a filled circle like this:



The PROG2.PRG program running

Summary of Lesson 2

- Again, 520ST/STE users cannot use a fully integrated system because there is not enough memory to support this - you must note down any errors/warnings and use the Goto line... (Alt-G) feature of the editor.

- 1Mb+ users have an fully-integrated system where the compiler tells the editor about errors and you can use the Jump to Error (Alt-J) command on the Program menu to find all the errors/warnings.

- A sidenote - the warning about argument type incorrect would not have been given by older, non-ANSI, compilers. Lattice C 5 knows about function prototyping and is able to check the types of function arguments.

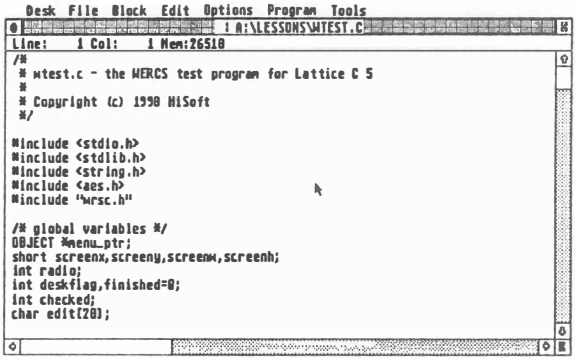
If you are wide awake you may have noticed that the compiler reported two errors (look back at the output on page 13) and that we only corrected one of them to obtain a successful compilation. This is because the second error was caused by the first error; the compiler became confused as to the meaning of the program and thought that a semi-colon was overdue. This is an example of a spurious error caused by a previous problem - always be on the look out for this type of error.

Lesson 3 - Optionally yours

In this lesson we are not going to differentiate between 512K and 1Mb+ users and we shall simply tell you to compile, link and run your program - please refer to Lesson 1 if you are unsure how to do this.

Run EDC.PRG (512K users) or LC.PRG (1Mb+ users), insert your working disk and type Alt-L or select Load from the File menu

Single-click on the Lessons directory to open it and then double-click on WTEST.C to load it into the editor.

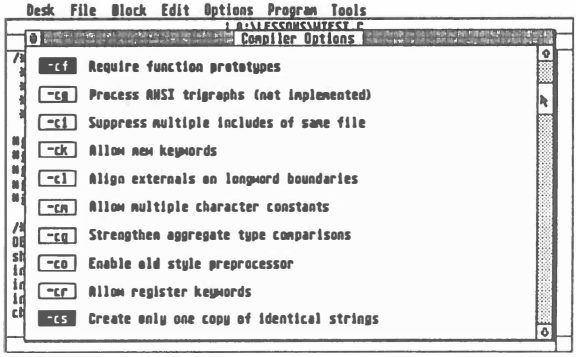


The WTEST.C Program

This program is an extended example of using the GEM system and uses structures created with the WERCS resource editor - see the chapter **WERCS, The Resource Editor** for more details.

Our concern here is to show you how to compile it using some useful compiler options; these are used to let you modify the way the compiler behaves when compiling your program.

To see most of the compiler options available to you, select Compiler Options... on the Options menu or type Control-O, from within the editor. The following list will appear:



The Compiler Options list

As you can see, from scrolling down the list using the slider on the right hand side, there are a great many such options giving you enormous flexibility in your compilation. The options that are enabled by default are shown highlighted in black.

The options we are going to use in this program are:

-csf this is a multiple option, the -c is a prefix which can be followed by certain other letters to give particular options.

- -cs *create only one copy of identical strings* - this keeps the program smaller by only keeping one copy of any identical constant strings.
- -cf *require function prototypes* - function prototypes were introduced into the C language by the recent ANSI standard and are used to tell the compiler exactly how a function should be called, thus making it easier for you. Lattice C 5 checks for a function prototype and, if this option is used, will warn you if a prototype is absent.

Thus -csf says *create only one copy of identical strings* and *require function prototypes*; you can add other -c options onto this list.

-v *disable stack checking code* - this option tells the compiler not to generate any inline code in your program that will check that there is sufficient stack space during your program's execution. You would normally only use this option when you had a completely finished program that had been fully tested and that you wished to be of a minimum size and to run at maximum speed.

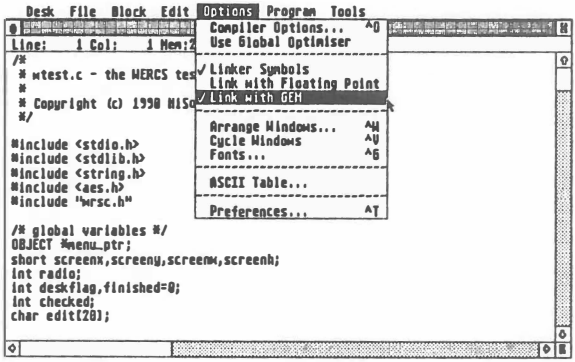
-Lag this is a multiple option, the -L is a prefix which can be followed by certain other letters to give particular options for the linker.

-La *add symbols to the created program, for debugging purposes* - this adds all your global variable and function names to your program so that a debugger can pick them up and make it easier for you to debug your program.

-Lg *link with GEM* - this ensures that the linker searches the GEM libraries when resolving references on your program.

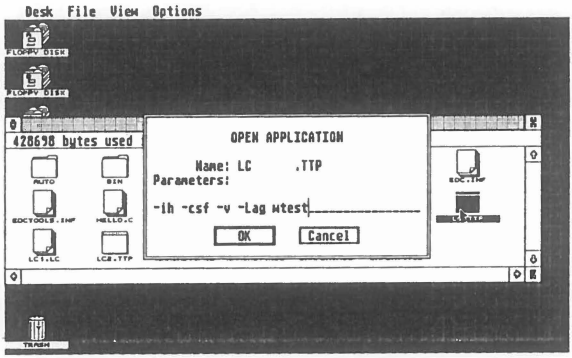
Note that both these options are available from the Options menu when compiling interactively, from the editor. When using LC.TTP you would use them on the command line, instead.

So, if you are running interactively, type Control-O and select -cs, -cf & -v and then go up to the Options menu and ensure that it looks like:



The Editor's Options menu

Otherwise, if you have less than 1Mb of memory, quit the editor, invoke LC.TTP as in Lesson 1 and type the following command line:



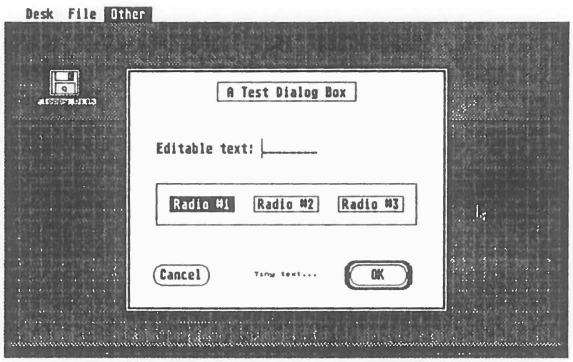
512K Users - The Command Line for WTEST.C

Compile and link the program as usual - you will create WTEST.PRG.

1Mb+ users can do this, and run the result, simply using Alt-X.

512K owners will use LC.TTP as described and then double-click on WTEST.PRG from the desktop.

Either way, the running program looks like:



WTEST.PRG running

Summary of Lesson 3

- The Lattice C 5 system is extremely flexible and allows the user great freedom of choice.

- There are many compiler and linker options that affect how these tools will treat your program. These options also often affect your program's size and speed of execution; it is wise to use them carefully and only when you need them.

- It is very easy to create GEM programs with Lattice C 5; once you have experimented a little you might like to read through the WERCS chapter later in this manual which contains much invaluable information regarding GEM. **Volume III** (the Atari Library manual) is worth dipping into for further information on the various aspects of GEM.

That completes our brief but, we hope, useful introduction to using the Lattice C 5 system. We now encourage you to get on and use the package, referring to this and the other volumes as and when you need to although you might like to glance at the final section in this chapter before you do.

Hints and Tips

Here is some advice on getting the best out of your Lattice C 5 system and your Atari ST.

- Always read the error messages very carefully - they invariably provide much information about the source of the problem. Do not expect the error message to necessarily occur in the line that is actually at fault, the compiler may often report an error one line too late. Also, the error pointer (^) often points after the error, not at it.

- If you are totally lost as to why your program is not working, use the manuals in the first instance - look up the library function definition and check that you are using it correctly and including the right header files. The manuals are full of useful and relevant information, please use them for reference as often as possible.

- It is best if you do not ignore warnings generated by the compiler - it spots many semantic mistakes and gives you guidance, through the warnings, as to errors you may be making in your usage of types.

  In particular, the argument type incorrect warning means that you are passing an expression to a function that is not consistent with what the function expects - *do not just cast the argument* (force its type) to get yourself out of trouble, either change the definition of the function you are calling or change the type of the variable to be appropriate to that expected by the function. See Lesson 2 above.

- If (when!) your program crashes don't be afraid to use the debugger - it is worth getting used to single-stepping and setting breakpoints, not only will you gain understanding of how your program is running but you will also speed up the discovery of obscure runtime errors. See the chapter on MonST2C for more detail.

- If possible, use a RAM disk for the intermediate (quad) files generated by the compiler since this has a massive effect on the speed of compilation. Do this by setting the QUAD environment variable - see the chapter on **LC, the Compiler**.

- If the linker reports undefined symbols and they are not your program's fault, make sure that the first file loaded by the linker is c.o (or the equivalent file required by the library that you are using) i.e. put it first in the linker's command. If you are using more than one library then lc.lib (or the equivalent) should be searched last.

- If you wish to create a linker control (with) file then don't get put off by the many options, get the LC.TTP driver to build it for you by using the -L option, see the **LC, the Compiler** chapter.

- All the advanced features of the Lattice linker (CLink) naturally require that you use Lattice format objects rather than GST format objects. It is always worth changing any existing assembly language code to be able to be assembled by asm so that you may use CLink.

- If you have code that does not use prototypes, perhaps because you haven't used an ANSI compiler before, you can use the -pr option of the compiler to generate a prototype file for your functions, automatically.

- You can turn stack checking on and off for each individual module in a multi-module program. This can be particularly useful if one module of a program is highly recursive.

- If your program has several large static arrays, the best code will be produced if you use the default (small, -b1) data model i.e. don't use the -b0 option. Instead, declare the large arrays as far, explicitly.

- Unlike most 68000 C compilers, Lattice C 5 lets you produce extremely large programs that use the small, fast branch instructions of the 68000 instruction set. Thanks to the advanced *alv* (automatic link vector) facility of the linker these branches are extended when required. The -r0 option overrides this feature making all branches long, for GST compatibility - unless you have a very good reason to do otherwise, do not use the -r0 option; we have never used -r0, even though LC.PRG is nearly 200K long and is compiled with itself.

- To generate small, fast programs, assuming that you are using prototypes, use the -rr option which will cause up to four parameters to be passed in *registers* rather than on the stack which is slower and requires more code. Note that this option may not be used if you are *not* using prototypes.

- Using 16 bit integers as the default (via the -w flag) will give better code than using the default 32 bit integers. However we recommend strongly that you use prototypes with this option, since otherwise it is easy to end up with the wrong number of bytes being placed on the stack when calling a function, often with disastrous results!

- As always, the more memory you have, the easier and the faster will be your development. Memory prices are reasonably low at the time of writing and we would advise you to make sure you have at least one megabyte of memory in your machine as soon as possible!

- You will also find that the purchase of a hard disk drive will revolutionise your attitude to program development, as long as you organise it well and keep regular backups!

Right, now it's up to you!

EdC
The Screen Editor

Introduction

This chapter details the use of the editor, EdC, and how to invoke other parts of the system from it; it does not detail those tools themselves. EdC is an enhanced version of the editor supplied with HiSoft DevpacST and HiSoft BASIC. In many ways EdC is more than an editor, it is a visually-orientated shell that will let you run almost the whole Lattice C system from within it using a single keystroke or menu click. Having said that you will need either a hard disk or two double-sided floppies to take full advantage of this.

EdC comes in two versions, one that includes the compiler (LC.PRG) and the other that does not (EDC.PRG). They are both used in the same way, but the version with the compiler has extra commands, and uses more memory of course. In the rest of this section EdC refers to those facilities that are available from both EdC and LC; whilst LC refers to those features that are only available from LC.PRG.

To run EdC, double-click on the EDC.PRG icon from the Desktop or type EDC from Batcher. LC.PRG is loaded in the same way.

When the editor has loaded, a menu bar will appear and an empty window will open, ready for you to enter your programs.

The Editor

The editor section of EdC is a 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 manipulate blocks of text. 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 use most of the commands from the keyboard without having to touch the 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. If you have enough memory you can edit a file of over 300k (though make sure your disk is large enough to cope with saving it if you do!). As all editing operations, including operations like searching, are RAM-based they act very quickly.

When you have typed in your program you must be able to save it 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. It will also let you edit up to four files at once, so that you can check the contents of a header file whilst writing your program.

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 easy and obvious to remember, while the Ctrl commands are based on those used in WordStar, and many other compatible editors since.

Entering text and Moving about

Having loaded EdC, 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 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 subsequently when you press a key.

Cursor keys

To move the cursor around the text to correct errors or to enter new characters, you use the cursor keys, labelled ← → ↑ and ↓. If you move the cursor past the right-hand end of the line this won't add anything to your text, but if you try to 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 necessary.

If 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.

For those of you used to WordStar style editors, the keys Ctrl-S, Ctrl-D, Ctrl-E and Ctrl-X work in the same way as the cursor keys.

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 Ctrl-R or Shift ↑. To move the cursor a page down, you can click on the lower grey part of the scroll bar, or press Ctrl-C or Shift ↓.

If you want to move the cursor to a specific position on the screen you may move the mouse pointer to the required place and click (there is no WordStar equivalent for this feature!).

Tab key

Pressing the Tab key inserts a special character (ASCII code 9) into your text, 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.

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, and only one byte on disk, but can show on screen as many more, allowing you to tabulate your program neatly, without increasing its size unduly. You can change the tab size before or after loading EdC using the Preferences command described shortly.

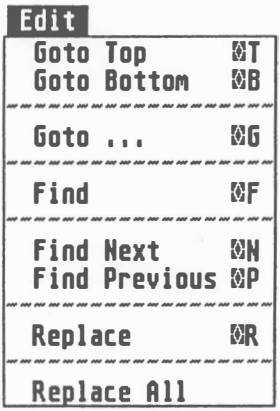
Backspace key

Pressing 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 at 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 commands on the Edit menu may also be used to move the cursor about your text:



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.

Go to top of file

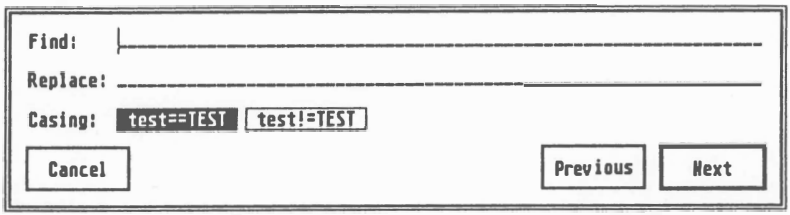
To move to the top of the text, click on Goto Top from the Edit menu, or press Alt-T. The screen will be re-drawn if necessary starting from line 1.

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.

Searching and Replacing Text

To find a particular section of text click on Find from the Search menu, or press Alt-F. A dialog box will appear,



This allows you to enter the find and replace strings. In the example above long has been entered as the find string and int 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 at that point 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 not stop if LONG was found; if test==Test was selected then the search would find LONG.

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 Edit 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 Edit 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.

Deleting 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. (This is equivalent to the WordStar sequence Ctrl-Q Y).

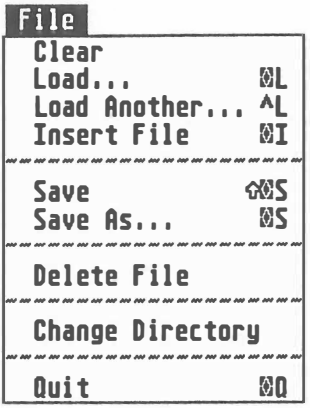
UnDelete Line

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 over individual lines.

Delete all text

To clear out the current text, click on Clear from the File menu. If you have made any changes to the text that have not been saved onto disk, a confirmation is required and an alert box will appear. Click on OK to delete the text, or on Cancel to abort the operation. If you wish to save the text before clearing the buffer, click on the Save button.

Disk Operations



Save As...

To save the text you are editing, click on Save As... from the File menu, or press Alt-S. The GEM 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 an error occurs a dialog will appear showing a TOS error number, the exact meaning of which can be found under _OSERR in **Volume II - Library manual.**

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 Backups are selected from the Preferences options 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), EdC 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.

Loading Text

To load in a new text file, click on Load from the File menu, or press Alt-L. If you have made any changes that have not been saved, a confirmation will be required. The GEM 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 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 then 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.

If you wish to continue editing the current file and would like to edit another file then use Load Another... or press Ctrl-L. This will open the file in the next unused window.

When loading EdC from Batcher, or any other CLI, you may include up to four 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.

Inserting Text

If you want 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 GEM 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

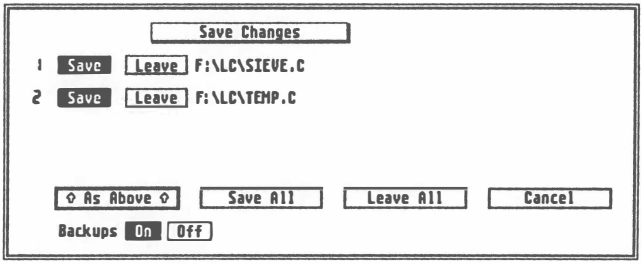
If you want to delete a file on disk (if for instance you have run out of disk whilst trying to save), click on Delete File. The GEM file selector will appear, allowing you to select a suitable disk and filename. Clicking OK or pressing Return will then delete the file from the disk. If an error occurs a dialog will appear showing a TOS error number, the exact meaning of which can be found under _OSERR in **Volume II - Library manual**. If you click on Cancel the file will *not* be deleted.

Change Directory

This option allows you to move the current directory; 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 GEM 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 will not be changed.

Quitting EdC

To leave EdC, 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, like this:

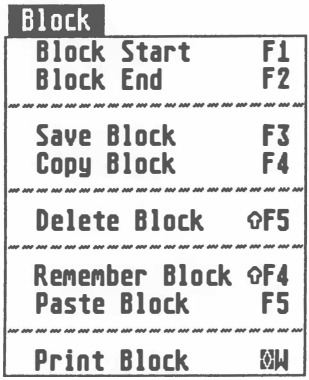


This example shows that two files have changed. Clicking on Save All, As Above or pressing Return will exit the editor saving the changes. Clicking on Cancel will return to the editor. Leave all will ignore all the changes you have made.

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 F:\LC\TEMP.C in the above example and then pressed Return, only the F:\LC\SIEVE.C file 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.

Block Commands



A *block* is a marked section of text which may be copied to another section, deleted, printed or saved onto disk. The function keys are used to control blocks.

Marking a block

The start of a block is marked by moving the cursor to the required place and selecting Block Start or pressing key F1. The end of a block is 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.

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.

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 block buffer, for later use.

Copy block to block buffer

The current marked block may be copied to theblock 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.

Pasting a block

A block in the block buffer may be pasted at the current cursor position by clicking on Paste Block or by pressing F5.

The block buffer will be lost if the edit buffer size is changed.

Printing a block

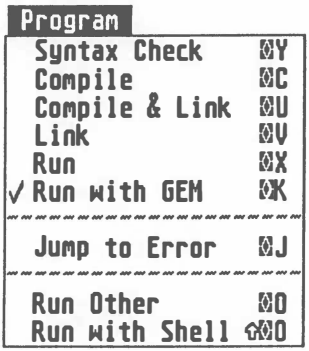
A marked block may be sent to the printer by clicking on Print Block or by pressing Alt-W. An alert box will appear confirming the operation and clicking on OK will print the block. The printer port used will depend on the port chosen with the Install Printer desk accessory, or will default to the parallel 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.

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

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

Compiling & Running Programs

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



Syntax Check

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

Compile

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

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

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

Jump to Error

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

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

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

Link

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

Compile & Link

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

Run

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

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

Run with GEM

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

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

Run Other...

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

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

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

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

```
%?.o
```

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

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

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

Run with Shell

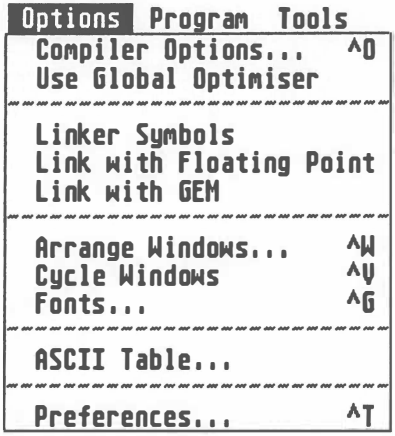
This command is available from both versions of EdC. It lets you run other programs from within the editor, then return to it when they finish. The keyboard shortcut for this command is Shift-Alt-O.

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

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

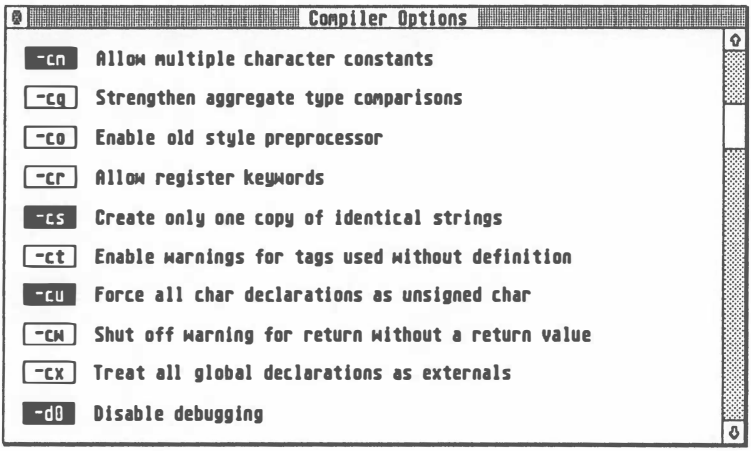
The same expansion of the current filename as used by Run Other can be used by this command. If you wish to use the same command more than once you will probably save time by using the Tools menu.

Options



Compiler Options

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



Use Global Optimiser

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

Linker Symbols

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

Link with Floating Point

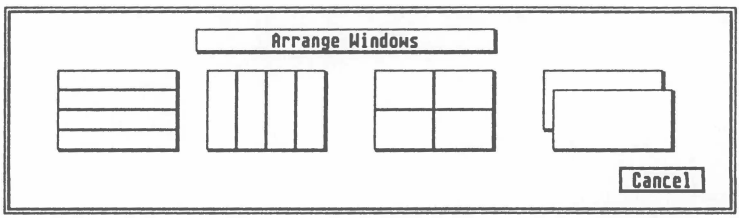
You should select this option if you are using the linker interactively and your program uses floating point as it causes your program to be linked with the floating point maths library.

Link with GEM

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

Arrange Windows...

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



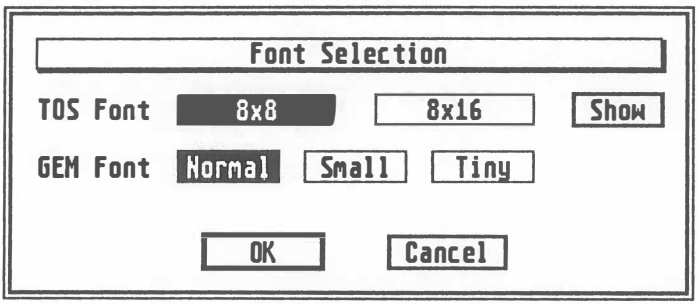
Clicking on the Cancel button will leave the windows arranged as they were before the Arrange Windows command was invoked.

Cycle Windows

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

Fonts...

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



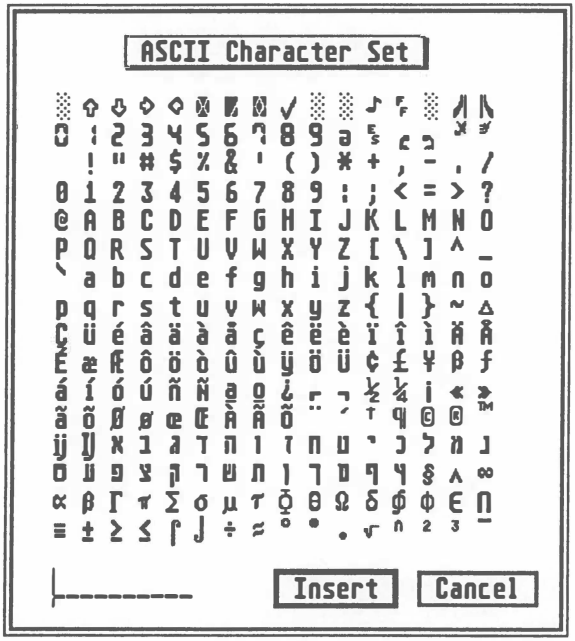
The GEM Font is the font that will be used by the editor to display text. In monochrome there are three fonts available as above. Changing to Small will double the number of line displayed on the screen to 40. With the Tiny font the characters are only 6 pixels by 6 pixels wide but this does mean that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

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

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

ASCII Table...

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

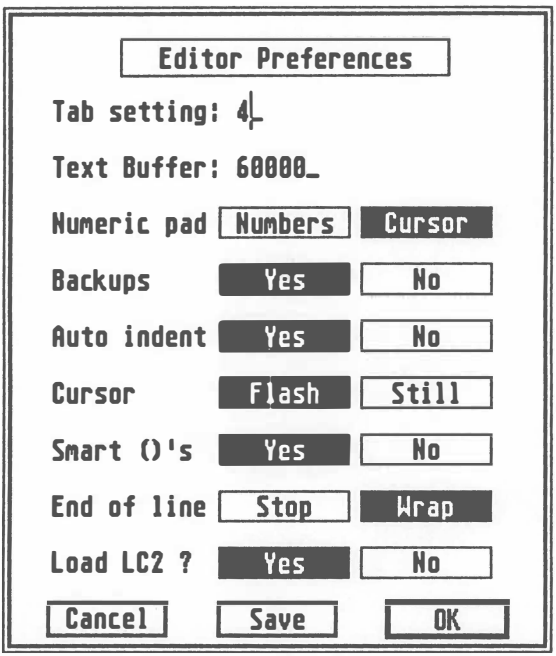


You may click on individual characters and they will be shown in the line at the bottom of the box. Pressing Return or clicking on the Insert button will then add the characters to the text that you are editing at the current cursor position.

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

Preferences...

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



Tabs

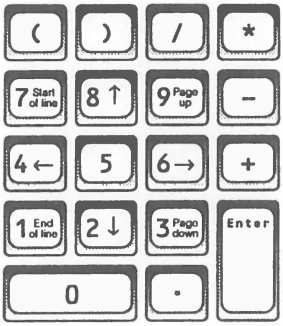
By default, the tab setting is 4, but this may be changed to any value from 2 to 16.

Text Buffer

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

Numeric pad

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



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

Backups

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

Auto indenting

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

Cursor

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

Smart ()s

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

End of Line

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

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

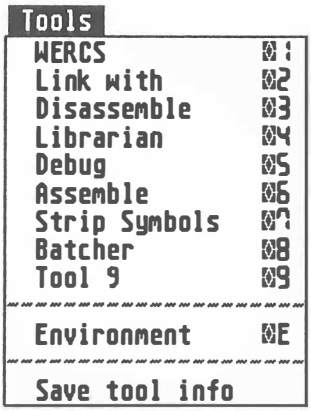
Load LC2

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

Saving Preferences

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

Tools Menu

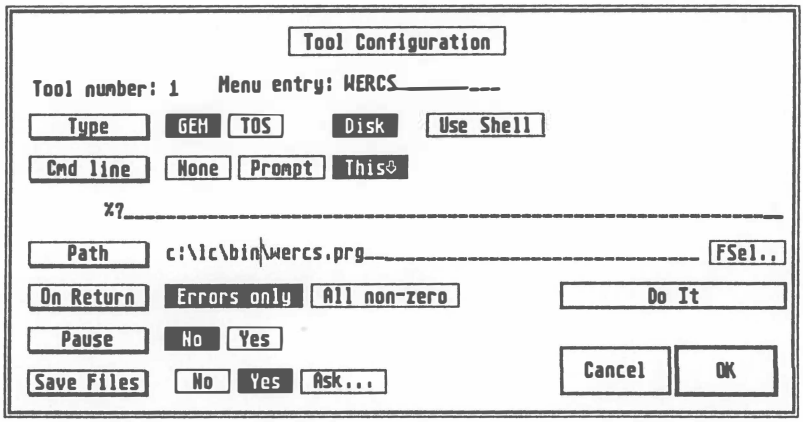


The Tools menu lets you run programs of your choice from within the editor using a single keystroke or click of the mouse. To take full advantage of this facility you will need at least one megabyte of RAM and either a hard disk or two double-sided floppies.

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

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

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



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

Menu entry

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

Type

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

Disk/Use Shell

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

Cmd line

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

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

On Return

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

Pause

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

Save Files

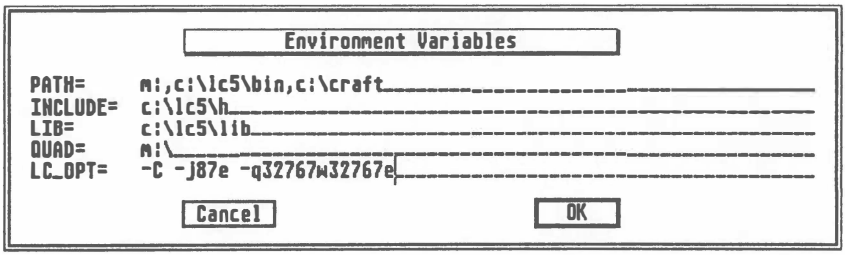
These options change which files will be saved before running the tool. If you select No then no files will be saved, selecting Yes (the default) will save *all* files (not just the current window), whilst Ask... will prompt you using the Save/Leave dialog described under Quitting EdC.

Running Tools

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

Environment

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



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

Save tool info

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

Miscellaneous Commands

About EdC

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

Help Screen

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

Switching Windows

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

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

Windows & Desk Accessories

The Editor Windows

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

Desk Accessories

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

Automatic Double-Clicking

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

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

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

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

Saved! Desk Accessory Users

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

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

LC
The Compiler

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

The Compiler Driver

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

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

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

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

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

Return Codes

The lc command returns the following completion codes:

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

1 One or more fatal compilation errors were reported.

2 No source files were found.

The Compiler Phases

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

The Parser and Pre-processor

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

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

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

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

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

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

The code generator

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

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

The global optimiser

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

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

Register assignment

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

Dead store elimination

Stores of values which are never fetched again are eliminated.

Dead code elimination

Code whose value is not used is eliminated.

Global common sub-expression merging

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

Hoisting of invariants out of loops

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

Induction variable transformations

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

Copy propagation

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

Constant propagation and folding

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

Auto variable elimination and re-mapping

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

Very busy expression hoisting

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

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

the expression i+j will be computed in only one place.

Various reductions in strength

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

Various arithmetic operations involving constants are reduced in strength.

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

Various control flow transformations

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

Reordering to reduce value lifetimes

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

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

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

Environment Variables

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

Setting the variables

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

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

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

The environment variables recognised and used by the compiler are:

PATH — Executable path

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

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

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

INCLUDE — Include path

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

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

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

LIB — Library path

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

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

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

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

QUAD — Quad path

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

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

```
QUAD=m:\
```

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

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

LC_OPT — Default options

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

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

Pre-processor Symbols

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

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

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

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

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

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

Compiler Options

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

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

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

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

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

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

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

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

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

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

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

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

c Allows comments to be nested.

d Allows $ character to be used in identifiers.

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

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

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

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

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

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

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

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

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

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

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

```
struct XYZ *p;
```

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

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

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

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

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

-d0 Disables all debugging information.

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

-d Same as -d1.

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

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

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

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

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

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

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

```
#define symbol
```

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

```
#define symbol value
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

s Toggles listing of the input source code.

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

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

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

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

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

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

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

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

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

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

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

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

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

Several messages may be affected with the same -J option such as -J22l30e132w which disables warning message 22, turns 30 into an error and enables 132 as a warning.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
clink WITH xxx.lnk
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

p Causes the compiler to generate prototypes with __PROTO for portability to other compilers.

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

Note that -pres will not generate any prototypes.

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

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

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

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

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

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

-q3w
Quit after 3 warnings or errors.

-q2e
Quit after 2 errors.

-q10w1e
Quit after 10 warnings or any errors.

-q

Quit after any errors or warnings.

-q-

Never quit on any errors or warnings.

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

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

0 Defaults all subroutine calls to far which means that the compiler will use an absolute 32-bit relocated address to locate the target function. Note that any functions explicitly declared near will use the more efficient 16-bit relative offset.

1 The compiler default, causes all subroutine calls to be defaulted to near which means that the compiler will use a 16-bit PC relative address to locate the target function. In order for this to work, the target subroutine must be within +/-32K of the generated instruction. If it is not within range, the linker will generate an ALV to allow the call to be bridged to the final target. Any functions explicitly declared far will use the larger 32-bit address.

r Causes the compiler to use registerised parameters for all subroutine calls and entry points. The first two integral and two pointer items will be loaded into d0-d1/a0-a1 for the call. Any function without a prototype or explicitly declared __stdargs will use the normal stack conventions.

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

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

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

```
-Rmylib.lib
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Pre-compiled Header Files

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

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

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

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

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

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

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

Language Extensions

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

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

const

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

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

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

enum

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

For example, this statement defines an enum type:

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

and this defines some objects of that type:

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

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

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

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

signed

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

void

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

```
void john(int x);

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

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

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

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

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

```
void *p;

if (*p)
  ...
```

will generate error 29, invalid pointer operation.

volatile

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

Storage Classes

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

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

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

far

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

huge

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

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

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

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

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

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

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

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

Calling Conventions

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

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

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

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

__asm

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

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

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

__interrupt

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

__regargs

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

__saveds

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

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

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

__stdargs

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

Built-in Functions

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

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

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

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

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

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

```
#undef strlen
```

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

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

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

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

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

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

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

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

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

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

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

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

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

Inline Calls

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

The directive has the form:

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

The various parts of the directive are:

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

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

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

where:

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

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

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

```
#define __TRAP_1    "4e41"

void _vgs(short,short);

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

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

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

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

```
#define __LINEA_D   "a00d"

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

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

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

The #pragma inline directive may also be used to extend the language to encompass features of the processor which the language cannot express. Consider a general rotate instruction:

```
#define __ROL_L_D1_D0    "e3b8"

unsigned long _lrotl(unsigned long,int);

#pragma inline d0=_lrotl(d0,d1)    { __ROL_L_D1_D0; }
```

This would then allow the use of the 68000 rotate instruction in the instruction stream without recourse to a function.

A more complex function might be used to pack two short word values into a long word. This might have the form:

```
unsigned long pack(short,short);

#define __SWAP_D0          "4840"
#define __MOVE_W_D1_D0     "3001"

#pragma inline d0=pack(d0,d1)
  { __SWAP_D0; __MOVE_W_D1_D0;}
```

Compiler Operational Errors

These indicate that the compiler is having trouble operating correctly because it cannot access required files or cannot obtain enough disk or memory space. Some of these errors are caused by not providing the required prototyping information to phase 1 so that it will manipulate a function into a format usable by phase 2, or by some other misuse of a low level facility.

-i option ignored

More than 16 -I option strings were specified. Only the first 16 are retained and used.

-r option has been moved to LC1

A -r option was detected on phase 2. This option was moved to the first phase of the compiler with the version 5 release.

__builtin_fpc requires -f8 switch on LC1

A call has been made to __builtin_fpc during phase 2 when phase 1 was not using the MC68881 flag. The first phase should be re-run with the -f8 flag.

Argument to abs must be an integral type

In phase 2 the function __builtin_abs has been passed a non-integral type. This function must be prototyped correctly so that the first phase will convert any parameters to the required type.

Argument to emit must be an integral constant

In phase 2 the opcode to __builtin_emit is a non-constant value. Only constant values may be used for the opcode specification.

Arguments to max/min must be an integral type

In phase 2 the function __builtin_min or __builtin_max has been passed a non-integral type. These functions must be prototyped correctly so that the first phase will convert them to the required type.

Can't create debugger intermediate file

The first phase of the compiler could not create its intermediate file for the debugging output. This error usually results from a full directory on the output disk.

Can't define __LINE__ and __FILE__

An attempt has been made to pre-define __LINE__ or __FILE__ from the command line; such redefinitions are illegal.

Can't open debugging file

The second phase could not open the debugging symbol file.

Can't open precompiled header file

One of the pre-compiled header files specified via the -H option could not be opened by the first phase.

Can't create object file

The second phase of the compiler was unable to create the object file. This error usually results from a full directory on the output disk.

Can't create quad file

The first phase of the compiler was unable to create the quad file. This error usually results from a full directory on the output disk, or an attempt to use a nonexistent disk.

Can't open file for pre-processor output

The first phase of the compiler was unable to open the pre-processor output file. This error usually results from a full directory on the output disk.

Can't open prototype file

The first phase of the compiler was unable to open the prototype file. This error usually results from a full directory on the output disk.

Can't open quad file

The second phase of the compiler was unable to open the quad file. This error usually occurs when you call phase 2 of the compiler (lc2) directly with an invalid quad file name.

Can't open source file

The first phase of the compiler was unable to open the source file. This error usually occurs because you mis-spelt the file name or did not specify the proper drive and/or directory path.

Can't open symbol file

The first phase of the compiler was unable to open the symbol file. This error usually results from a full directory on the output disk.

Combined output file name too large

The output file name constructed by combining the source or quad file name with the text specified using the -o option exceeded the maximum file name size of 64 bytes.

Corresponding message not found in lc1.lc

The error number which the compiler generated could not be found in the error file lc1.lc.

Dead assignment eliminated <symbol>

The global optimiser has detected a redundant assignment to symbol. Note that these assignments may not be visible in your program, but generated internally by the compiler.

End of file on object file

The second phase of the compiler detected an end of file condition on the object file. This usually indicates a full disk.

Error message too long in lc1.lc

An error message in the lc1.lc file exceeded the maximum length permitted.

Error reading symbol file

The second phase encountered an error when reading the debugging symbol file.

File name missing

The source file name was not specified.

File name too large

The name of the file passed to the second phase exceeded the maximum file name length.

File too short

A pre-compiled header file expired prematurely. This indicates that the structure of the pre-compiled header file is damaged in some way.

Floating point opcode must be a constant

In phase 2 the opcode to __builtin_fpc is a non-constant value. Only constant values may be used for the opcode specification.

Full path name of source file too long -- not retained

The full pathname exceeded the maximum length permitted in the debugging output file and so was not retained.

Intermediate file error

The first phase of the compiler encountered an error when writing to the quad file. This error usually results from an out-of-space condition on the output disk.

Invalid -b option

The character following the -b option was not 0 or 1; see the section **LC, The Compiler** for the valid options.

Invalid -e option

The character following the -e was not a 0, 1, or 2. This usually occurs because the line was mistyped. Retype the line and try again. See the section **LC, The Compiler** for a list of the valid compiler control options.

Invalid -f option

One of the characters following the -f option was not a recognised compiler control character. See the section **LC, The Compiler** for a list of the valid compiler control options.

Invalid -j option. Warning/error not specified

Following the -j option to lc1 an error number was not specified; see the section **LC, The Compiler** for details of the -j option.

Invalid -m option

One of the characters following the -m option was not a recognised compiler control character. See the section **LC, The Compiler** for a list of the valid compiler control options.

Invalid -r option

One of the characters following the -r option was not a recognised compiler control character. See the section **LC, The Compiler** for a list of the valid compiler control options.

Invalid -s option

One of the characters following the -s option was not a recognised compiler control character. See the section **LC, The Compiler** for a list of the valid compiler control options.

Invalid attribute flags for builtin function

An __builtin function has the wrong attribute types; this indicates that it was incorrectly prototyped during the first phase. Check the types of the parameters in the prototype.

Invalid intermediate file

During the second phase the intermediate file was found to be damaged in some manner.

Invalid option

An invalid command line option was specified, and that option will be ignored. See the section **LC, The Compiler** for a list of the valid compiler control options.

Invalid register specification for getreg

A register specification to __builtin_getreg was invalid. The register specification must be a constant in the range 0 to 15.

Invalid register specification for putreg

A register specification to __builtin_putreg was invalid. The register specification must be a constant in the range 0 to 15.

Invalid symbol definition

The name attached to -d specifying a symbol to be defined was not a valid C identifier or was followed by text which did not begin with an equals sign.

lc1.lc is corrupt

An error occurred whilst the compiler was attempting to read the error file lc1.lc.

No functions or data defined

The compiler reached the end of the source file without finding any data or function definitions. One common cause of this error is to forget a comment terminator (*/) during the first function in the source file. This causes the compiler to gobble up your program as if it were a comment.

No register specified for ASM call

During phase 2 a function has been declared using an __asm convention, but no register was supplied where one was needed.

Not enough memory

This message is generated when either phase of the compiler uses up all the available working memory.

Parameters beyond file name ignored

Additional information was present on the command line beyond the name of the source file. A common source of this error is to place compiler options after the file name.

Reference has overlapping definition <symbol>

The global optimiser has detected a reference to symbol which overlaps a definition which it has seen. This may or may not indicate an error in your program, but should be investigated.

Same register used twice for parameters

During phase 2 a register has been used twice in an __asm call. All registers used in such a call must be distinct.

Seek error on object file

The second phase of the compiler detected a seek error on the object file. This usually indicates a full disk.

Symbol file corrupted

The second phase detected a corrupt symbol file. This rarely occurs and may be related to a full disk or lack of disk integrity.

Unable to continue compilation due to previous BLTN error(s)

The second phase has been forced to abort due to previous errors encountered when processing __builtin functions.

Unable to find 'lc1.lc'

The first phase is unable to find the error file lc1.lc which contains its error messages.

Undefined variable <symbol>

The global optimiser has detected a reference to symbol without seeing any definition of this symbol. This almost certainly indicates a bug in your program.

Unrecognized -c option

One of the characters following the -c option was not a recognised compiler control character. See the section **LC, The Compiler** for a list of the valid compiler control options.

Unrecognized -j option

One of the characters following the -f option was not a recognised compiler control character. See the section **LC, The Compiler** for a list of the valid compiler control options.

Unrecognized -p option

One of the characters following the -p option was not a recognised compiler control character. See the section **LC, The Compiler** for a list of the valid compiler control options.

Unrecognized -q option

One of the characters following the -q option was not a recognised compiler control character. See the section **LC, The Compiler** for a list of the valid compiler control options.

Value for putreg must be an integral type

In phase 2 the function __builtin_putreg has been passed a non-integral type. This function must be prototyped correctly so that the first phase will convert them to the required type.

Wrong number of parameters for builtin function

In phase 2 an __builtin function has been passed the wrong number of parameters.

Syntax errors and warnings

These indicate that the compiler is having difficulty understanding your C source program. The message includes the source file name and line number identifying the point at which the problem was detected. An *error message* indicates that the problem prevents the construction of a usable object module and must, therefore, be corrected. A *warning message* indicates that the compiler detected something unusual but will proceed to make an object module, using appropriate assumptions about what you intended the source code to do.

Syntax errors and warnings are reported via a message with the following format:

```
fff nnn Error xxx: mmm
```

where the message components are:

fff	This is the name of the source file that was being processed when the error occurred.
nnn	This is the number of the source file line that was being scanned when the error occurred. Source file lines begin at 1, not 0.
xxx	This is the error number, as listed below.
mmm	This is the error message text.

All messages listed below indicate *fatal errors* unless the message number listed below is followed by (W). When a fatal error occurs, the compiler will not produce a usable object module. The lc command alerts you to this condition by beeping and pausing, unless you use the -C option to force continuous compilation.

If the message number below is followed by (W), then it is a *warning*. When such a message is displayed, the compiler will produce a usable object module by making reasonable assumptions about what you intended the source file to do. Nonetheless, it's a good idea to investigate these warnings, since the compiler's assumptions may disagree with your intentions.

1 invalid pre-processor command

This error is generated by a variety of conditions in connection with pre-processor commands, including specifying an unrecognised command, failure to include white space between command elements, or use of an illegal pre-processor symbol.

2 unexpected end of file

The end of an input file was encountered when the compiler expected more data. This may occur on an #Include file or the original source file. In many cases, correction of a previous error will eliminate this one.

3 file not found <name>

The file <name> specified on a #Include command was not found.

4 invalid lexical token

An unrecognised element was encountered in the input file that could not be classified as any of the valid lexical constructs (such as an identifier or one of the valid expression operators). This may occur if control characters or other illegal characters were detected in the source file.

5 invalid macro usage

A pre-processor #define macro was used with the wrong number of arguments.

6 line buffer overflow

Expansion of a #define macro caused the compiler's line buffer to overflow. This may occur if more than one lengthy macro appeared on a single input line or if the closing parenthesis was missing from a macro invocation.

7 file stack full

The maximum extent of #Include file nesting was exceeded; the compiler supports #Include nesting to a maximum depth of 16.

8 invalid conversion

A cast (type conversion) operator was incorrectly specified in an expression.

9 undefined identifier <name>

The named identifier was undefined in the context in which it appeared; that is, it had not been previously declared. This message is only generated once; subsequent encounters with the identifier assume that it is of type int (which may cause other errors).

10 invalid subscript expression

An error was detected in the expression following the [character (presumably a subscript expression). This may occur if the expression in brackets was empty.

11 string too large or not terminated

The length of a string constant exceeded the maximum allowed by the compiler (256 bytes). This will occur if the closing " (double quote) was omitted in specifying the string.

12 invalid structure reference

The expression preceding the period (.) or indirect (->) structure reference operator was not a structure or pointer to a structure.

13 member name missing

An identifier indicating the desired aggregate member was not found following the period (.) or arrow (->) operator.

14 undefined member <name>

The indicated identifier was not a member of the structure or union to which the period (.) or arrow (->) referred.

15 invalid function call

The identifier preceding the left parenthesis of a function call was not implicitly or explicitly declared as a function.

16 invalid function argument

A function argument expression following the left parenthesis on a function call was invalid. This may occur if an argument expression was omitted.

17 too many operands

During expression evaluation, the end of an expression was encountered but more than one operand was still awaiting evaluation. This may occur if an expression contained an incorrectly specified operation.

18 unresolved operator

During expression evaluation, the end of an expression was encountered but an operator was still pending evaluation. This may occur if an operand was omitted for a binary operation.

19 unbalanced parentheses

The number of opening and closing parentheses in an expression was not equal. This error message may also occur if a macro was poorly specified or improperly used.

20 invalid constant expression

An expression which did not evaluate to a constant was encountered in a context which required a constant result. This may occur if one of the operators not valid for constant expressions was present.

21 illegal use of aggregate

An identifier declared as a structure or union was encountered in an expression where aggregates are not permitted. Only the direct assignment and conditional operators may be used on aggregates, and explicit or implicit testing of aggregates as a whole is not permitted.

22 (W) structure used as function argument

An identifier declared as a structure or union appeared as a function argument without the preceding & operator. Aggregates may be passed by value, so this is a legal construction. The warning message is generated to alert you that earlier versions of Lattice C (before version 3) passed the address of the aggregate in this case.

23 invalid use of conditional operator

The conditional operator was used erroneously. This may occur if the ? operator was present but the : was not found when expected.

24 pointer operand required

The context of the expression required an operand to be a pointer. This may occur if the expression following * did not evaluate to a pointer.

25 modifiable lvalue required

The context of the expression required an operand to be an lvalue. This may occur if the expression following & was not an lvalue, or if the left side of an assignment expression was not an lvalue.

26 arithmetic operand required

The context of the expression required an operand to be arithmetic (not a pointer, function, or aggregate).

27 arithmetic or pointer operand required

The context of the expression required an operand to be either arithmetic or a pointer. This may occur for the logical OR and logical AND operators.

28 missing operand

During expression evaluation, the end of an expression was encountered but not enough operands were available for evaluation. This may occur if a binary operation was improperly specified.

29 invalid pointer operation

An operation was specified which was invalid for pointer operands (such as one of the arithmetic operations other than addition).

30 (W) pointers do not point to same type of object

In an assignment statement defining a value for a pointer variable, the expression on the right side of the = operator did not evaluate to a pointer of the exact same type as the pointer variable being assigned, i.e. it did not point to the same type of object. The warning also occurs when a pointer of any type is assigned to an arithmetic object. Note that the same message may be a fatal error if generated for an initialiser expression or in some situations involving mixed memory models.

31 integral operand required

The context of an expression required an operand to be integral, i.e. one of the integer types (char, Int, short, unsigned, or long).

32 invalid conversion specified

The expression specifying the type name for a cast (conversion) operation or a sizeof expression was invalid.

34 invalid initialiser expression

The expression used to initialise an object was invalid. This may occur for a variety of reasons, including failure to separate elements in an initialiser list with commas or specification of an expression which did not evaluate to a constant. Some experimentation may be required in order to determine the exact cause of the error.

35 closing brace expected

During processing of an initialiser list or a structure or union member declaration list, the compiler expected a closing right brace, but did not find it. This may occur if too many elements were specified in an initialiser expression list or if a structure member was improperly declared.

36 (W) control cannot reach this statement

A statement within the body of a switch statement was not preceded by a case or default prefix which would allow control to reach that statement. This may occur if a break or return statement is followed by any other statement without an intervening case or default prefix.

37 duplicate statement label <label>

The specified statement label was encountered more than once during processing of the current function.

38 unbalanced braces

In a body of compound statements, the number of opening left braces { and closing right braces } was not equal. This may occur if the compiler got out of phase due to a previous error.

39 invalid use of keyword <keyword>

One of the C language reserved words appeared in an invalid context (e.g. as a variable name).

40 break not inside loop or switch

A break statement was detected that was not within the scope of a while, do, for, or switch statement. This may occur due to an error in a preceding statement.

41 case not inside switch

A case prefix was encountered outside the scope of a switch statement. This may occur due to an error in a preceding statement.

42 invalid case expression

The expression defining a case value did not evaluate to an integral constant.

43 duplicate case value

A case prefix was encountered which defined a constant value already used in a previous case prefix within the same switch statement.

44 continue not inside loop

A continue statement was detected that was not within the scope of a while, do, or for loop. This may occur due to an error in a preceding statement.

45 default not inside switch

A default prefix was encountered outside the scope of a switch statement. This may occur due to an error in a preceding statement.

46 more than one default

A default prefix was encountered within the scope of a switch statement in which a preceding default prefix had already been encountered.

47 while missing from do statement

Following the body of a do statement, the while clause was expected but not found. This may occur due to an error within the body of the do statement.

48 invalid while expression

The expression defining the looping condition in a while or do loop was null (not present). Indefinite loops must supply the constant 1, if that is what is intended.

49 else not associated with if

An else keyword was detected that was not within the scope of a preceding if statement. This may occur due to an error in a preceding statement.

50 label missing from goto

A statement label following the goto keyword was expected but not found.

51 label name conflict <label>

The indicated identifier, which appeared in a goto statement as a statement label, was already defined as a variable within the scope of the current function.

52 invalid if expression

The expression following the if keyword was null (not present).

53 invalid return expression

The expression following the return keyword could not be legally converted to the type of the value returned by the function.

54 invalid switch expression

The expression defining the value for a switch statement did not define an integral value or a value that could be legally converted to an integer.

55 (W) no case values for switch statement

The statement defining the body of a switch statement did not contain at least one case prefix.

56 (W) colon expected

The compiler expected but did not find a colon (:). This error message may be generated if a case expression was improperly specified, or if the colon was simply omitted following a label or prefix to a statement.

57 (W) semi-colon expected

The compiler expected but did not find a semi-colon (;). This error generally means that the compiler completed the processing of an expression but did not find a statement terminator. This may occur if too many closing parentheses were included or if an expression was otherwise incorrectly formed. Because the compiler scans through white space to look for the semi-colon, the line number for this error message may be beyond the actual line where a semi-colon was needed.

58(W) missing parenthesis

A parenthesis required by the syntax of the current statement was expected but was not found (as in a while or for loop). This may occur if the enclosed expression was incorrectly specified, causing the compiler to end the expression early.

59 invalid storage class

In processing declarations, the compiler encountered a storage class invalid for that declaration context (such as auto or register for external objects). This may occur if, due to preceding errors, the compiler began processing portions of the body of a function as if they were external definitions.

60 incompatible aggregate types

The types of the aggregates involved in an assignment or conditional operation were not exactly the same. This error may also be generated for enum objects, which are treated as integers.

61 (W) undefined struct/union tag <name>

The indicated structure or union tag was not previously defined; that is, the members of the aggregate were unknown. Note that a reference to an undefined tag is permitted if the object being declared is a pointer, but not if it is an actual instance of an aggregate. This message may be issued as a warning after the entire source file has been processed if a pointer was declared with a tag that was never defined.

62 structure/union type mismatch

A structure or union tag has been detected in the opposite usage from which it was originally declared (i.e., a tag originally applied to a struct has appeared on an aggregate with the union specifier). The Lattice compiler defines only one class of identifiers for both structure and union tags.

63 (W) duplicate declaration of item <name>

The indicated identifier has been declared more than once within the same scope. This error may be generated due to a preceding error, but is generally the result of improper declarations.

64 structure contains no members

A declaration of the members of a structure or union did not contain at least one member name.

65 invalid function definition

An attempt was made to define a function body when the compiler was not processing external definitions. This may occur if a preceding error caused the compiler to get out of phase with respect to declarations in the source file.

66 (W) invalid array limit expression

The expression defining the size of a subscript in an array declaration did not evaluate to a positive integral constant. This may also occur if a zero length was specified for an inner (i.e. not the leftmost) subscript of an array object.

67 illegal object

A declaration specified an illegal object as defined by this version of C. Illegal objects include functions which return arrays and arrays of functions.

68 illegal object for structure

A structure or union declaration included an object declared as a function. This is illegal, although an aggregate may contain a pointer to a function.

69 structure includes instance of self

The structure or union whose declaration was just processed contains an instance of itself, which is illegal. This may be generated if the * is forgotten on a structure pointer declaration, or if (due to some intertwining of structure definitions) the structure actually contains an instance of itself.

70 invalid use of structure qualifier

The formal parameter of a function was declared illegally as a function.

71 formal declaration error <name>

A variable was declared before the opening brace of a function, but it did not appear in the list of formal names enclosed in parentheses following the function name.

72 external item attribute mismatch

An external item has been declared with attributes which conflict with a previous declaration. This may occur if a function was used earlier, as an implicit int function, and was then declared as returning some other kind of value. Functions which return a type other than int must be declared before they are used so that the compiler is aware of the type of the function value.

73 (W) declaration expected

In processing the declaration of objects, the compiler expected to find another line of declarations but did not, in fact, find one. This error may be generated if a preceding error caused the compiler to get out of phase with respect to declarations.

74 (W) initialiser data truncated

A string constant used as an initialiser for a char array defined more characters than the specified array length. Only as many characters as are needed to define the entire array are taken from the first characters of the string constant.

75 invalid sizeof expression

An attempt was made to apply the sizeof operator to a bit field, which is illegal.

76 left brace expected

The compiler expected, but did not find, an opening left brace in the current context. This may occur if the opening brace was omitted on a list of initialiser expressions for an aggregate.

77 identifier expected

In processing a declaration, the compiler expected to find an identifier which was to be declared. This may occur if the prefixes to an identifier in a declaration (parentheses and asterisks) are improperly specified, or if a sequence of declarations is listed incorrectly.

78 undefined statement label <label>

The given statement label was referred to in the most recent function in a goto statement, but no definition of the label was found in that function.

79 (W) duplicate enumeration value

More than one identifier within the list for an enumeration type had the same value. While this is not technically an error, it is usually of questionable value.

80 invalid bit field

The number of bits specified for a bit field was invalid. Note that the compiler does not accept bit fields which are exactly the length of a machine word (such as 32 on a 32-bit machine); these must be declared as ordinary int or unsigned variables.

81 pre-processor symbol loop (macro expansion too long or circular)

The current line contains a reference to a pre-processor symbol that is a circular definition.

82 maximum object/storage size exceeded

The size of an object exceeded the maximum legal size for objects in its storage class; or, the last object declared caused the total size of declared objects for that storage class to exceed that maximum.

83 (W) reference beyond object size

An indirect pointer reference (usually a subscripted expression) used an address beyond the size of the object used as a base for the address calculation. This generally occurs when an expression makes reference to an element beyond the end of an array.

84 (W) redefinition of pre-processor symbol <name>

A #define statement was encountered for an already defined symbol. The first definition is pushed, so that an additional #undef statement is needed to undefine the symbol.

85 (W) function return value mismatch

The expression specifying the value to be returned by a function was not of the same type as the function itself. The value specified is automatically converted to the appropriate type; the warning merely serves as notification of the conversion. The warning can be eliminated by using a cast operator to force the return value to the function type. This warning is also issued when a return statement with a null expression (i.e. no return value) appears in a function which was not declared void; generation of the warning for this particular context can be disabled using the -cw option on the lc command.

86 (W) formal definitions conflict with type list

The types of the formal parameters declared in the actual definition of a function did not agree with those of a preceding declaration of that function with argument type specifiers.

87 (W) argument count incorrect

The number of function arguments supplied to a function did not agree with the number of arguments in its declaration using argument type specifiers.

88 (W) argument type incorrect

The type of a function argument expression did not agree with its corresponding type declared in the list of argument type specifiers for that function.

89 (W) constant converted to required type

The type of a constant expression used as a function argument did not agree with its corresponding type declared in the list of argument type specifiers for that function.

90 invalid argument type specifier

The type specifier for an argument type in a function declaration was incorrectly formed. Argument type specifiers are formed according to the rules for type names in cast operators or sizeof expressions.

91 illegal void operand

One of the operands in an expression was of type void; this is expressly disallowed, since void represents no value. This is often caused by attempting to assign the result of a function declared as 'void' returning to a variable.

92 (W) statement has no effect

An expression statement did not cause either an assignment or a function call to take place. Such a statement serves no useful purpose, and can be eliminated; usually, this error is generated for incorrectly specified expressions in which an assignment operator was omitted or mistyped.

93 (W) no reference to identifier <name>

An object with local scope was declared but never referenced within that scope. This warning is provided as a convenience to warn of declarations that may no longer be needed (if, for example, the code in which the variable was used was eliminated but not its declaration). It may also occur if the only use of the object is confined to statements which are not compiled because of conditional compilation directives such as #Ifdef or #If.

94 (W) uninitialised auto variable <name>

An auto variable was used in an expression without having been previously initialised by an assignment statement or appearing in a function argument list with a preceding & (i.e. its address passed to a function). Note that the compiler considers the variable initialised after any statement causes it to be initialised, even though control may not flow from that statement to other subsequent uses of the variable. Note also that this warning will be issued if the third expression in a for statement uses a variable which has not yet been initialised, which may be incorrect if that variable is initialised inside the body of the for statement.

95 (W) unrecognised #pragma operand

The operands of the #pragma statement did not match the syntax expected. This may occur, for instance, if an Inline directive had missing semi-colons.

99 (W) attempt to change a const lvalue

The program is writing to a const object. This is not permitted by the definition of such objects.

100 (W) no prototype declared for function

A function was called with no in-scope prototype. This message can only occur if the -Cf option is set.

101 redundant keywords in declaration

More keywords than permitted appeared in the declaration of a variable or function.

102 conflicting keywords in declaration

The attempted declaration contained conflicting keywords (e.g. near far).

103 (W) uninitialised constant <name>

A variable has been declared with the const modifier, but no initialiser has been supplied so that the variable is, by definition, undefined and cannot subsequently be initialised (since it is constant).

104 (W) conversion from pointer to const/volatile to pointer to non-const/volatile

A conversion from a pointer to an object specified using the const or volatile modifier has been assigned to a pointer which does not have that property. Hence the compiler will be unable to honour the const/volatile property of the object when accessed through the new pointer.

106 postfix expression not allowed on a constant

An attempt was made to use a postfix operator (++, etc.) on a constant.

107 too many initialisers

The declaration encountered contained more initialisers than elements existed to place the values into.

109 Invalid use of type name or keyword

The type name used was illegal in the context in which it occurred, e.g. attempting to pass a typedefd name as a parameter to a function.

116 (W) Undefined enum tag <name>

The program is accessing an object of type enum with the named tag, which has not been defined.

117 Enum contains no members

An empty enumeration declaration has occurred. This will happen if the tag space between the left and right brace is empty.

118 Conflicting use of enum/struct/union tag <name>

A structure, union or enumeration tag has been detected in a different usage from which it was originally declared (e.g. a tag originally applied to a struct has appeared on an aggregate with the union specifier). The Lattice compiler defines only one class of identifiers for structure, union and enum tags.

119 Identifiers missing from definition of function <name>

An attempt is being made to define a function using the prototyped format, however a name has not been supplied for one of the parameters.

121 (W) Hex constant too large for char (high bits will be lost)

The hexadecimal or octal constant just defined is too large to fit into an object of type char. This will occur if one is defining a constant larger than 127. The -cm option of lc1 may be used to permit multi-character constants.

122 (W) Missing ellipsis

An ellipsis (...) did not appear in a function prototype when the compiler expected to see one (e.g. after a trailing comma).

123 No tag defined for enumeration (cannot construct prototype)

The enumeration type used in a function prototype currently has no tags defined, hence the compiler cannot construct a function prototype to represent it.

124 Debugger symbol table overflow

The amount of information required to generate the source level debugging output has overflowed the compiler's table.

125 Invalid number

The 'number' encountered cannot be parsed as a legal number.

126 (W) #endif, #else, or #elif out of order

The compiler has detected a #endif, #else or #elif without a matching #if.

127 Operand to # operator must be a macro argument

An attempt has been made to use the stringisation operator (#) on an identifier which did not appear as a macro argument in a macro definition.

128 #error usage

This error number is allocated to the ANSI #error directive. The error message issued will be that specified by the user.

132 (W) Extra tokens after valid preprocessor directive

A valid pre-processor directive was parsed (e.g. #endif), but tokens which could not be attached to the directive that appeared afterwards.

133 Cannot redefine macro <name>

An illegal attempt is being made to redefine a macro. This will occur if you attempt to define any of the predefined macros (__FILE__ etc.).

135 (W) Assignment to shorter data type (precision may be lost)

An assignment has been made from a wider data type to a more narrow one (e.g. long to short). This may cause data loss at run time.

136 Invalid use of register keyword

The register keyword cannot be used in the context it appeared in.

139 (W) Missing #endif

The end of compilation unit was reached whilst a #endif was still pending.

140 (W) sizeof operator used on array that has been converted to pointer

The sizeof operator has been applied to an array which was converted quietly to a pointer type. This occurs when a function argument has array limits of [] i.e. is of indeterminate size.

142 (W) Array size never given for <name>

The compilation unit finished before the named array's size had been defined. This will occur if a tentative definition occurred in the module (of the form, for instance, char x[];)

143 Object has no address

An attempt has been made to obtain the address of a register variable which can have no address. Note that this error is issued whether or not the object is actually assigned to a register.

144 Combined storage for strings and constants exceeds maximum

The size of the static near data area exceeds the maximum of 64K. This error would be issued at link time if not issued by the compiler. Note that there is no limit on far data.

154 (W) no prototype declared for function pointer

The function pointer being accessed has not been fully specified so that a complete prototype is not known for it (e.g. void (*x)() would have no prototype, whilst void (*x)(void) would).

155 (W) no statement after label

A label appears, but no statement appears after it. The standard requires that a statement appears after every label, even if it is the null statement (;).

Internal Errors

These indicate that the compiler encountered some internal condition that should not have occurred. They are reported via the message:

```
CXERR: xx
```

where xx is the error number. When such a message occurs, compilation is terminated immediately, and both the quad file and the object file are probably unusable. If you get one please send us copies of the source files you are attempting to compile.

1	Invalid error or warning message code number.
2	Call to function not applicable to the current environment.
3	Invalid symbol table access.
4	Declaration chain is broken.
5	An unlink error occurred while processing an "undef".
6	The compiler attempted to push back too many tokens.
7	There is no aggregate list for a structure reference.
8	Stack underflow has occurred.
9	Invalid attempt to generate the address of a constant.
10	A test value is not a constant.
11	Invalid unary operator.
12	Invalid binary operator.
13	A scaling object is not a pointer or array.
14	Unexpected end-of-chain while restoring internal context.
15	Invalid quad type.
16	Deletion length is less than two bytes.
17	Insufficient memory.
18	An error occurred when releasing memory.
19	Invalid condition during temporary assignment.

20	Invalid condition while processing program section.
21	Literal pool generation error.
22	Invalid condition while processing data section.
23	Invalid quad file.
24	End-of-file while processing "for" quad.
25	Invalid register number.
26	Temporary save or restore error.
27	Invalid operand size.
28	Invalid storage base.
29	Error during branch folding.
30	Error during control statement processing.
31	Error during special addressing setup.
32	Invalid object description block offset.
33	Too many function parameters.
34	Indirect argument for call-by-reference.
35	Invalid external relocation value.
36	Error during search of the debugging information lists.
37	Error during search of library lists.
38	Array size invalid for optimisation
38	Error determining pointer size
39	Invalid register parameter quad
40	Unsupported type for register parameter
41	Invalid register parameter specification
42	Byte register required
77	Out of index registers
99	Miscellaneous error

CLink
The Linker

CLink is the standard linker for Lattice C and may either be used in the integrated mode as described in **EdC, The Screen Editor**, or directly from the command line.

The linker command line specifies which files are to be linked together and in what order. Note that the order of linking is significant as this allows a symbol defined in a module linked earlier to override one in a later module; this is often useful when replacing standard library routines with your own custom versions.

A simple CLink command line

To link a single C object file together with a startup stub and library the command line used could be:

```
CLINK c.o mine.o LIB lc.lib
```

this will produce an executable program (assuming no errors occur) named mine.prg; note that the name of the executable is taken from the second named file in the link sequence.

Concepts

CLink provides several unusual features whilst linking, this allows more flexibility when initially writing your program leaving many of the decisions up to the linker.

ALVs

When CLink is collecting all of 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 and 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.

Near DATA/BSS

CLink supports a 64K near data section which can be accessed via a global base register (traditionally A4). This section is formed from all sections which are named __MERGED (as described in the assembler section) and then several variables are created by the linker to allow the initial base of this to be set up. This is discussed later under the **Reserved symbols** section.

Directives

The CLink directives allow the input files and the format of the output file to be specified.

Input directives

The input directives allow the names of the object files to be linked to be passed to the linker. The linker works by collecting all sections which have identical types into a single output section; note that apart from the special name __MERGED section names are ignored when generating executable files.

When a file is required by CLink it initially looks in the current directory for the file, if it is found there then that file is used, otherwise a search is made for it in the paths mentioned in the LIB environment variable. The LIB variable consists of a list of semi-colon (;) or comma (,) separated items which indicate paths where the file should be searched for, e.g.

```
LIB=c:\lc\lib;c:\mylibs
```

FROM files

Specifies the object files that are the input files for the linker. If the first item on the command line is a filename then the FROM keyword is optional and may be omitted. FROM may be used more than once with the files for each FROM adding to the list of files to be linked.

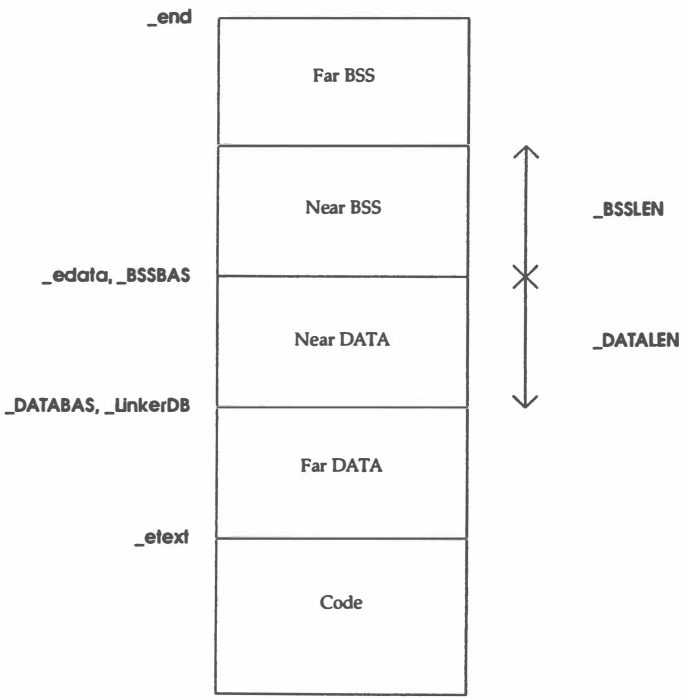
To specify more than one file in a single FROM statement they may either be listed after it separated by spaces or +, e.g.

```
FROM a.o+b.o
```

LIB files Specifies the files to be scanned as libraries. Only modules within the library which contain symbols which are referenced will be included in the final object module. Note that LIBRARY is a synonym for LIB. The same syntax used for specifying multiple FROM files may be used for multiple libraries.

Output directives

The output directives control the format and type of the final file created by the linker when a link has been completed successfully. The output file generated by the linker is normally directly executable by GEMDOS (unless the PRELINK option has been used) and is, by default, named the same as the second object module supplied by a FROM option with its .o suffix replaced by .prg; if only one file is linked then the the output name will be based on that file. The format of the file is identical to the normal GEMDOS executable file, but with the DATA and BSS sections split so that they contain both the Near (__MERGED) data and Near (__MERGED) BSS sections. This leads to a load map of the following form:



Note that the symbols marked are all discussed under the **Reserved symbols** section below.

ADDSYM This causes CLink to emit standard DRI symbols for all symbols in the input object files regardless of whether the input object file was compiled with one of the -d options. Note that the option XADDSYM is normally preferred.

NODEBUG Suppresses any symbol table information or symbolic debug information in the final object file. This is equivalent to the object file that would be produced if Strip were run on the final object file. Note that ND is a synonym for NODEBUG.

PRELINK Causes CLink to output an object module with references and definitions still intact so that it can be linked later on to produce a final executable file. This is designed for development of large projects where the programmer is only changing a single source module. Note that a prelinked object file cannot have ALVs inserted into it and so CLink may be unable to satisfy all 16 bit PC-relative references when linking with the prelinked file.

TO file Specifies the name of the output file which is to be created, overriding the default name generation discussed above. If the file name specified begins with a period (.) then the normal default name generation is performed using the extension specified after the period rather than .PRG.

XADDSYM This causes CLink to emit HiSoft extended symbols for all symbols in the input object files regardless of whether the input object file was compiled with one of the -d options. This is extremely useful for use with a symbolic debugger such as MonSTC. Note that that when the PRELINK directive is in force then XADDSYM is a synonym for ADDSYM.

Pre-linking

Pre-linking is similar to a normal link, however instead of producing the five load sections from identically typed sections, it coalesces only identically typed *and named* sections into *output sections*. If a section is unnamed then it is merged with the first named section of the same type. Note that the special name __MERGED is considered a type-modifier and hence only sections named __MERGED will be coalesced with __MERGED sections.

When pre-linking ALVs are applied according to the normal rules, i.e ALVs will be generated for out of range branches within a section, and *all* cross-output-section references.

During pre-linking variables will often have undefined values (since the modules in which these are defined are to be linked later) and so all of these variables are reported. Note that this means that an error has technically occurred and the return code from CLink will be non-zero. This is likely to be important to users of make type utilities.

Map files

A map file is a file describing the order and location of files and variables processed by the linker written to a normal file for perusal by the user. These files provide a large number of options for the programmer to customise the output format; they are enabled using the MAP directive which has the format:

```
MAP [[filename],option,option,...]
```

The filename gives the name of the file to which the map file is to be written, this may be of the form .MAP to indicate that the filename should be based on the output file name. The options specify which parts of the map file are to be written and all consist of single letters:

Option	Meaning
F	Produce a mapping of input files in the output file
H	Show where the input hunks (sections) were placed
L	Map the library placements
S	Show all external symbols
X	Show a cross reference of external symbols

When generating cross-reference information it is often useful to be able to separate this from the map file information. This can be done using the XREF directive which allows a separate cross-reference file to be specified. It has the form:

```
XREF filename
```

To control the layout of the map file several directives are available which are used in the same way as the more normal output directives or options:

FWIDTH n	Width of file names (default 16).
HEIGHT n	Lines on a page in map file, 0 indicates no pagination (default 55).
HWIDTH n	Width of hunk names (default 8).
INDENT n	Columns to indent on a line. This is included in width (default 0).
PWIDTH n	Width of program unit names (default 8).
SWIDTH n	Width of symbol names (default 8).
WIDTH n	Sets the maximum line length for the map and cross reference listings. This is useful when sending the output to a device which has different line length requirements. If not specified one width defaults to 80.

'WITH' files

A WITH file provides a method for encapsulating long and complex (or short and simple) CLink command lines in a control file, known as a WITH file, traditionally with the extension .LNK. The format of a WITH file is identical to the normal command line driven structure, except than line breaks may be used in place of spaces. For example the first simple command line example could have the WITH file:

```
FROM c.o mine.o
LIB lc.lib
```

Consider a slightly more complex example of program which is to consist of two modules (object files) and a library:

```
FROM c.o+mine.o+hers.o  ; the object files
LIB lc.lib              ; and a simple library
TO prog1.ttp            ; output file name
XADDSYM                 ; add HiSoft extended symbols
VERBOSE                 ; output messages during linking
MAP .map,F,H,X          ; produce map file
```

Note the use of the ; to delimit comments in a WITH file. This file can then be passed for execution to the linker using the command line (assuming the WITH file is saved as mywith.lnk)

```
CLINK WITH mywith.lnk
```

A more complex example would be to consider a mixed language example with both C and assembler modules. Assuming that the main project was written in C, the normal C runtimes would have to be included, additionally it may be necessary to force some external data items defined in the assembly language to use the _ prefix required by C, this can be done using the DEFINE directive:

```
FROM c.o              ; C startup code
a.o+b.o               ; assembler object files
d.o+e.o               ; C object files
LIB lcg.lib+lc.lib    ; GEM & C runtime libraries
TO prog2.prg          ; output file
DEFINE _menu=menu     ; alias menu defined in assembly
                      ; to menu referenced in C
MAP .map,f,h,l,s      ; map file
XREF .xrf             ; separate cross-reference file
HEIGHT 66             ; longer page length
FWIDTH 10             ; narrower filename width
```

Note that the contents of WITH files are always processed *after* any files explicitly named on the command line, hence if the last WITH file were named mywith2.lnk, then the command line:

```
CLINK WITH mywith2.lnk LIB mylib.lib
```

would search the mylib.lib file *before* searching the lcg.lib or lc.lib files.

Compiler Options and CLink

The -b1 option

This option causes all data, within a program, to be merged into one large block which is referenced via 16-bit references. CLink is aware of where these references occur and will order the output so that the theoretical maximum of 64K is possible.

The -r1 option

The -r1 option causes all subroutine references to be via 16-bit relative branches. References which span more than 32k are noted by CLink and given an ALV, or *Automatic Link Vector*, which it creates, and are guaranteed to transfer control to the actual routine the caller was trying to reach.

The -d options

If a module is compiled with debugging turned on (any of -d1 through -d5), then CLink will parse this debugging information if it encounters any errors, and if a line number can be found, report the line number on which the error occurred. This is significantly more useful than merely reporting the module which caused the problem.

Reserved symbols

To provide access to the base of the sections created by the linker various symbols are invented by the linker. These are as follows:

__RESBASE, __RESLEN — Reserved symbols

These are reserved symbols used in the Lattice C resident startup code. If you use them in your own code your programs are almost certain not to work correctly.

__LinkerDB — Pointer to static merged data section

The address of this external variable points to the base of the merged data section. It is this variable which is referenced when a function is defined as __saveds.

__BSSBAS, __DATABAS — Base of merged data sections

These names refer to the base locations in the __MERGED data section. The location of __BSSBAS is the first byte of the merged BSS, whilst __DATABAS is the first byte of the merged data. These variables may be accessed using the code sequence (note that only one underscore is used since a further one is added automatically by the compiler):

```
extern void *far _BSSBAS,*far _DATABAS;
...
            void *x,*y;

            x=&_BSSBAS;
            y=&_DATABAS;
```

__BSSLEN, __DATALEN — Merged section lengths

The addresses of these names give the length of the respective __MERGED data section in *longwords*. Note that these variables *must* be accessed as longs otherwise the compiler may attempt to relocate them, giving random values as a result.

```
extern void *far _BSSLEN,*far _DATALEN;
...
            long a,b;

            a=(long)&_BSSLEN;
            b=(long)&_DATALEN;
```

__end, __edata, __etext — Last locations in program

These names refer to the last locations in the program. The address of _etext is the first location above the executable program text, that of _edata the first location above the initialised data area and _end the location immediately after the uninitialised data area.

Standard libraries

Because of the large number of options, a wide selection of libraries for use under the various compilation models is supplied. The 'normal' libraries are: c.o, the startup code; lc.lib the standard C library; lcm.lib the standard maths library; lcg.lib the standard GEM library.

All libraries are named according to consistent naming conventions made up of an initial leading letter, either c for a startup stub, or lc for a library, optionally followed by one of the following letters to indicate the type of the library:

Letter	Type of library
m	Library is for floating point maths
g	Library is for GEM support

Note that the standard C library must always be linked *after* the maths or GEM libraries but that the relative order of maths and GEM libraries is unimportant.

Following this prefix, none, one or more of the following letters are used to indicate the options used to compile the library:

Letter	Type of library
s	Default short integer library (-w)
r	Register parameter passing library (-rr)
nb	Non-base relative library (-b0)

The startup stubs may also be followed by an additional suffix; currently the suffices allocated are:

Suffix	Type of startup stub
acc	Desk Accessory
aut	Auto program type detecting
res	Resident program

Hence for instance the file lcsrnb.llb is a short integer, register passing non-base relative C library, whilst lcgr.llb would be a long integer register passing GEM library. Similarly a file called csrnbacc.o would be a short integer, register passing non-base relative Desk Accessory startup stub.

The GST libraries supplied follow identical naming conventions but use a .bln file extension to indicate that they are in the GST format.

CLink Messages

Whilst running CLink may discover things which it needs to bring to your attention. These may either be error messages or observations on the program which is being built.

CLink Warnings/messages

The messages in this section although warnings, will often indicate that the final program will be unusable in the form intended and you should not run it unless you are certain that you understand what you are doing.

Warning MERGED data > 64K, use -b0 on LC

The merged data section has exceeded the limit of 64K. The problem may be rectified by either compiling your program using the -b0 option on lc, or explicitly declaring some of your arrays far.

Warning! Absolute reference to <name> module: <mod> file: <file>

An absolute reference was detected to a merged data item, whilst building a resident load module. This warning will only be given if a reference has been made to the symbol __RESBASE, i.e. the linker is building a resident load module.

Warning: ALVs were generated

This message is generated when the NOALVS option is used, indicating that ALVs were generated. Note that this message will not be issued if the XNOALVS option is used.

Enter a DEFINE value for <name> (default __stub): Undefined symbols... First Referenced

These messages indicate that the linker has encountered a reference to a symbol for which it cannot locate a definition. The second message is issued if the BATCH keyword is specified, whereas the first allows you to specify an alternate name for the reference.

This error may occur because you have called a function using one name, but possibly mis-spelt the definition. Alternatively you may not be linking with the correct libraries, e.g. if the symbols mentioned include names such as _CXA55 then you are not linking with the floating point maths library, similarly, references to variables such as __AESpb indicate that you have used GEM, but not linked with the GEM library.

CLink Errors

These are the errors which may be issued by the linker. In general errors may be ignored by use of the IGNORE keyword to CLink, however programs so produced may not function correctly. The error numbers are broadly divided so that 200-400 may be issued by either pass. 401-500 are issued by pass 1, whilst 501-599 are issued during pass 2.

Note that if a module has been compiled with debugging turned on (-d1 through -d5) then the line number on (or near) to where the problem occurred will also be reported.

200 Out of memory!

The linker does not have enough memory left to successfully complete the link.

300 System error <val> on read

A system error occurred whilst attempting to read from the disk. This should only occur if the disk has been damaged in some way. The value of the error is given by <val>.

301 System error <val> on write

A system error occurred whilst attempting to write to the disk. This will normally indicate that the disk is full. The value of the error is given by <val>.

400 *** Break: Clink terminating

This message is printed when the operation of the linker is interrupted by the user pressing Ctrl-C. Note that the keyboard is only checked whilst screen input or output is occurring.

425 Cannot find library <file>

The file named in a LIB statement could not be located by the linker. This is probably due to a full pathname not being given for the file.

426 Cannot find object <file>

The file named in a FROM or ROOT statement could not be located by the linker. This is probably due to a full pathname not being given for the file.

443 '<file>' is an invalid file name

The filename specified in a FROM, LIB or ROOT statement is invalid. Typically this will be because the name is null.

444 hunk_symbol has bad <val> symbol <file>

A hunk_symbol hunk type was encountered by the linker which did not have the external type set to zero, but instead to val. If this error occurs it indicates that the named input file was damaged in some manner.

445 Invalid hunk_symbol <name>

A hunk_symbol hunk type was encountered by the linker during parsing of the external definitions. The named symbol was attached to this hunk.

446 Invalid symbol type <val> for <file>

Whilst parsing external declarations an unknown symbol type <val> was encountered in the named file.

448 <file> is not a valid object file

The named file did not match the specifications for an object module.

449 No hunk_end seen for <file>

On reaching the end of a hunk within the named file an end marker did not appear.

450 Object file <file> is an extended library

An attempt has been made to use a library as the operand of a FROM or ROOT statement. Libraries may only be searched, not included.

501 Invalid Reloc 8 or 16 reference

An attempt has been made to generate a branch between two differently named sections. Branches may only occur within a common section. This error will normally indicate an attempt to execute the data section!

502 <name> symbol - Distance for Reloc16 > 32768

The target of a 16 bit branch is more than 32K away from the reference. In general you should not see this message due to ALV generation.

503 <name> symbol - Distance for Reloc8 > 128

The target of an 8 bit branch is more than 128 bytes away from the reference. Note that the compiler does not generate such external branches and that the assembler does not allow their generation.

504 <name> symbol - Distance for Data Reloc16 > 32768

A 16 bit base-relative data section access is attempting to reach more than 32K. This error will normally indicate you are very close to the 64K limit on near data, and a a module has had its data section fall off the end of the merged data section (biassed by 32K). The solution is to reorder your modules putting the ones with large data sections alternatively you may have to move some of your near data to far.

505 <name> symbol - Distance for Data Reloc8 > 128

An 8 bit base-relative data section access is attempting to reach more than 128 bytes. This error will normally indicate incorrect code generation from the compiler.

506 Can't locate resolved symbol <name>

During the second pass the linker could not locate the named symbol in its table. This will either indicate an internal linker failure or a damaged library file.

507 Unknown Symbol type <val>, for symbol <name>

During the second pass the linker could not match the type of the named symbol in its table. This will indicate an internal linker failure.

508 Symbol type <val> unimplemented

Whilst parsing external declarations an unknown symbol type <val> was encountered in the named file. Note that the equivalent error (446) is reported during pass 1.

509 Unknown hunk type <val> in Pass2

The named file did not match the specifications for an object module. Note that this message is identical to the pass 1 error 448.

510 <name> symbol - Reference to unmerged data item

A module has attempted to access an unmerged (far) data item using a near access. This will usually occur if one module of a program is compiled using the -b1 option whilst another module uses -b0. The error message also suggest the action: try -b0 option on LC.

515 An ALV was generated pointing to data <name> symbol

An ALV was generated in the data section of the program. This will only occur if code generation has been performed in a data section, and as such this error will normally indicate an internal compiler failure.

600 Invalid command '<cmd>'

The named command was not recognised by the linker. The commands which are recognised are discussed in the section **CLink, The Linker**.

601 <cmd> option specified more than once

An attempt has been made to specify a command, which may only appear once, more than once, e.g. attempting to specify two TO files.

602 Unable to open output file '<file>'

The named output file could not be opened. This may be because the disk or directory is full.

603 <val> is not a valid number

The value <val> which appeared as a numeric argument could not be parsed as such.

604 with file is not readable

An error occurred whilst reading the WITH file.

605 Cannot open with file '<file>'

The named WITH file could not be opened.

607 No FROM files specified

No FROM or ROOT files were specified so the linker cannot start linking.

608 Premature EOF encountered

End-of-file occurred unexpectedly. This will normally indicate serious file system structure problems.

609 Error seeking in file <file>

An error occurred whilst attempting to seek about the named file. This will normally indicate serious file system structure problems.

611 Reloc found with odd address for symbol <name>, file <file>

A 16 or 32 bit relocation was attempted on a non word-aligned boundary. This is always illegal on the 68000.

ERROR: Invalid decimal constant '<val>'.

The value <val> which was entered in response to an undefined symbol was an invalid decimal constant.

ERROR: Invalid hex constant '<val>'.

The value <val> which was entered in response to an undefined symbol was an invalid hexadecimal constant.

ERROR: Multiply defined symbol '<name>'.

A symbol has been redefined. The file in which it first appears is named, as is the file in which the attempted re-definition occurs.

ERROR: Symbol '<name>' is not defined.

The named symbol which was entered in response to an undefined symbol was also undefined.

Hunk #n not written

The numbered hunk n was not written to disk. This will indicate an internal linker failure.

Unknown internal error

An internal error occurred whose error number was not recognised. This indicates a serious internal linker failure.

Batcher
The Command Shell

Batcher is a command line processor modelled on the COMMAND.COM processor of MSDOS. It can be used instead of the GEM icon interface if you like, or it can be used to run batch files. Unlike some other command line interpreters on the ST, you can still run GEM programs (such as EdC) from Batcher.

To run a program under Batcher, give the file name for the program. If you do not give an extension and a file without an extension cannot be found, Batcher will try to load a file with the extension .PRG, then .TOS and finally .TTP.

There are a number of commands built-in to Batcher i.e. no other program has to be loaded in order that they can be executed. These are described in the following sections.

Batcher can also be used to set up environment variable values used to configure the compiler, and to tell Batcher where to look for programs.

First we will describe the commands that are built in to Batcher and then we will discuss its line editing and batch file facilities.

AVAIL — Available memory

AVAIL displays the size of the largest free GEMDOS block of memory.

CD — Change Directory

The CD command changes the current directory. Directories are also known as *folders.*

This command takes one parameter; the name of the directory to go to. The syntax of the directory name follows the standard syntax used for path names. If the first character is a backslash (\) then the path is relative to the root directory. Otherwise, it is relative to the current directory. In addition, .. refers to the directory one level towards the root directory from the current directory.

Note that, following the operating system convention, there is a separate 'current' directory for each drive. This means that, to move to a particular directory, you should first use the Change Disk command described below and then use the CD command. For example to move to C:\LC you would use:

```
C:
CD \LC
```

This scheme has a useful advantage. Say you have two versions of a program in E:\MINE\OLD and D:\MINE\NEW. Then after using:

```
CD E:\MINE\OLD
```

and

```
CD D:\MINE\NEW
```

you can use E: and D: to refer to these two directories.

B: — Change Disk

B: (or b:) makes disk B the current disk. Any drive letter may be used so that whenever you reference a file without giving an explicit disk designator, the current disk will be used.

CLS — Clear screen

This command clears the screen and enables line wrap. If the COLOUR command has been used then the screen colours will be set as per the last colour command. CLS is useful of your program has left you with black text on a black background!

COLOUR — Set screen colours

This command takes two numeric parameters which set the background and foreground screen colours, using TOS rather than GEM colours, as shown in the table below

Number	High	Medium
0	White	White
1	Black	Red
2		Green
3		Black

e.g. If you are using mono-chrome:

```
COLOUR 0 1
```

would use black text on a white background. If you are using medium resolution:

```
COLOUR 3 1
```

would give red characters on a black background.

COPY — Copy Files

Copy files. You can copy single files or groups of files. You can use wildcards in the source and destination names. For example, to copy all the .C files from disk A to disk D enter the command:

```
COPY a:*.C D:
```

To make a backup code of all the .C files whose name starts with the letter F, you could use:

```
copy f*.c *.BAK
```

Note that the f is omitted in the second file specification.

You can ask to be prompted for the files to be copied by specifying the /a (for ask) flag. e.g.

```
COPY d:\myfiles\*.o a:\ /a
```

will prompt you with the names of all the .o files in the d:\myfiles directory.

You may reply:

Y	copy this file
N	ignore this file
Q or Ctrl-C	quit and don't copy any files
A	copy this and the remainder of the files

You can also modify the order in which the files are listed by using the /S (for size order) or /D (for date order) flags, optionally followed by a - to reverse the order; so

```
copy \fred\*.c a: /ad
```

would prompt you for the .c files in the fred directory on the root of the current drive, in date order.

Notice that you may enter commands and arguments in either upper or lower case.

Copy will use as much free memory as it can and read in as many files as possible. This means that copying files is often considerably faster than with the Desktop, especially on a single floppy system.

COPYWARN — Enable file overwrite warnings

The COPYWARN command lets you enable warnings when over-writing an existing file using the COPY command. Use

```
COPYWARN ON
```

to enable this. If you are prompted, press Y to over-write this file, N to leave this file and A to copy this and all subsequent files without asking.

DEL — Delete Files

DEL deletes files. You can use a wild card specification and Batcher will prompt you before it deletes each file to ensure that you really want to delete it. Type Y to delete this file; N not to delete it; Q to quit the delete command and A to delete the remaining files.

To avoid the prompting entirely, use the N flag:

```
DEL *.PRG/N
```

You can also modify the order in which the files are listed by using the /S (for size order) or /D (for date order) flags, optionally followed by a - to reverse the order.

DC — Disk Change

The ST's operating system can fail to notice that a floppy disk has been changed with the potentially disastrous consequence of corrupting the floppy. This is especially a problem when using disks with identical serial numbers under TOS 1.4 and 1.6.

The DC command can be used to ensure that the operating system notices that you have changed a floppy disk. It may be followed by a drive letter, but by default it will apply to drive A. Thus

```
DC
```

will ensure that the operating system notices that the disk in drive A has been changed and

```
DC B:
```

will inform the operating system that the disk in drive B has changed. If you find that use this command frequently, then you should consider using the DISKCHANGE command.

DIR — Directory List

This commands lists the files in a directory. You may follow DIR by a directory name or a drive name or a wild-card file name for which files to display. For example:

```
DIR headers
DIR B:
DIR d:\lc\*.c
```

If you specify the W flag, only the file names are displayed. Otherwise, the names, sizes and dates for the files are displayed.

```
DIR d:\sources\*.c/W
```

You can also modify the order in which the files are listed by using the /S (for size order) or /D (for date order) flags, optionally followed by a - to reverse the order:

```
DIR *.h/D
```

lists all the .H files in the current directory, in date order.

DISKCHANGE — Set auto-diskchange mode

Using

```
DISKCHANGE ON
```

causes Batcher to ensure that the operating system has noticed that a disk has changed whenever you use a built-in command that refers to a floppy disk drive. As such it takes a second or two before the command is executed. We thus recommend that it is not used on floppy only systems unless you are changing disks a great deal. Its effect can be disabled by using:

```
DISKCHANGE OFF
```

which is the default. You can ensure that the operating system notices a particular disk change by using the DC command, see above.

ECHO — Echo commands

Echo commands as they are processed. When echoing is on, each command in a batch file will be displayed on the screen before it is executed. You can use

```
ECHO OFF
```

to prevent the commands in batch files being echoed as they are performed and ECHO ON to turn this back on.

If the parameter to ECHO is not ON or OFF then the text will be echoed to the screen. For example:

```
ECHO This is a message
```

will display 'This is a message'.

ERA — Erase Files

This command is exactly the same as DEL and is supplied for the benefit of CP/M users.

EXIT — Exit Batcher

This command exits Batcher. You do not need to use this in .BAT files since Batcher will exit when it comes to the end of the batch file.

FORMAT — Format floppy disk

This will format standard ST double-sided disks without any interleave. It should be followed by the drive to format e.g.

```
FORMAT B:
```

will format the disk in drive B. A second (optional) parameter may be supplied giving the volume label which is to be given to the disk, e.g.

```
FORMAT B: VOLID
```

Formats a disk in drive B with a volume label of VOLID.

FREE — Free disk space

Returns the free space (in bytes) on a disk. For example,

```
FREE
```

gives the free space on the current disk while:

```
FREE A:
```

gives the free space on drive A.

MKDIR — Make Directory

This command creates a new directory (or folder) with the given name. You may use a full path specification if you wish, although MKDIR will not attempt to create more than one directory at once.

```
MKDIR SOURCES
```

will create a directory called SOURCES on the current disk.

MOUSE — Control Mouse Visibility

Turns the mouse on (use MOUSE ON) or off (MOUSE OFF). This is useful if you run a program which leaves the mouse in a funny state. Whenever you run a .PRG program, CLI enables the mouse. It disables the mouse on return. If a program disables the mouse and fails to re-enable it before it exits, subsequent programs will not show the mouse cursor.

Conversely, entering MOUSE OFF in Batcher will prevent the mouse cursor appearing when you run a program, which can be useful when running non-GEM applications that have extensions of .PRG.

When Batcher runs programs with extensions of .TOS and .TTP the mouse is not enabled.

To set the mouse back to the default value, issue MOUSE ON commands until the mouse appears and then issue MOUSE OFF commands until it vanishes again.

PAUSE — Pause for keypress

PAUSE will wait for a single key to be pressed. This can be used in batch files so that the user can read the previous output or change disks.

REM — Remark

Used to place a comment in a batch file - the line is ignored.

REN — Rename

Rename files. Wild cards can be used:

```
REN *.O *.OO
```

will rename all the files in the current directory with extension .O to have extension .OO.

RMDIR — Remove Directory

This command deletes a directory (or folder) with the given name. The directory must be empty before you can remove it. You may use a full path specification if you wish.

```
RMDIR A:\SOURCES
```

will delete a directory called SOURCES on the current disk if it does not contain any files.

SCREENSAVE — Set screensave mode

When turned on (SCREENSAVE ON) the screen is not cleared before a .PRG file is run. This is useful when running .PRG files that are not really GEM applications, so that you can see the output from previous commands.

SET — Set Environment Variable

Sets environment variables. If you just type SET, the current environment variable values are listed.

```
SET INCLUDE=d:\headers
SET PATH=d:\lc;c:\bin
```

You can remove a variable by setting it equal to no value. For example,

```
SET QUAD=
```

removes the QUAD environment variable.The environment variables used by Lattice C are described under **LC, The Compiler**.

SMALL — Set font size

This command is only really useful in monochrome.

```
SMALL ON
```

will cause TOS output, like that from Batcher, to appear in the 8x8 system font. On a high resolution screen, this gives 50 lines on the screen.

```
SMALL OFF
```

selects the 8x16 font giving the usual 25 lines in monochrome, but only 12 in medium resolution.

TYPE — Type File

TYPE displays a file on the screen.

```
TYPE HELLO.C
```

VIRTUALDISK — Perform virtual disking

This command is used by the installation program so that Batcher can perform 'virtual disking' itself rather than using the operating system routines, which do not work reliably. However, as all Batcher commands do not support this facility, we do not recommend that it is used interactively.

WHICH — Which file would run

This command returns the path of the file that will be run if you use a given filename. For example,

```
WHICH WERCS
```

will tell you from where WERCS will be loaded if you type WERCS to Batcher. This can be very useful if it appears that you are running the wrong program, particularly if it is called TEST !

Line Editing

When using Batcher you can also use the cursor keys. The ← and → keys will move the cursor within the current line, whilst you can us the ↑ key to display the previous command that you entered to Batcher. This is very useful if you have made a simple typing mistake. The area of memory where these commands are stored is known as the *history buffer*. You may press ↑ more than once; this will display the other lines that you entered previously. If you go too far back in the history buffer, then press ↓ and then commands will be displayed in the order in which they were entered.

Pressing Backspace will delete the character to the left of the cursor; Delete will delete the character under the cursor. Pressing Ctrl and ← will take you to the beginning of the current line and Ctrl and → will take you to the end of the line.

You can also recall the last line that starts with a particular character sequence by typing ^ first. For example:

```
^lc [Return]
```

will display the last line that started with LC and you can then edit this line if required. Using a prefix of ! will cause the last such command to be run. For example:

```
!cl[Return]
```

will probably run the last CLink command.

Batcher also supports some of the traditional MS-DOS line editing keys:

F1	copy one character from the previous line
F3	copy the rest of the previous line
F5	clear the current line
Delete	ignore one character from the previous line
Insert	cause the following characters to be inserted. Pressing Insert again switches this off.

The best way to learn about these features is to experiment.

Batch file facilities

Batch files are very useful for re-compiling groups of files. For example, you might have a single batch file that re-compiles (and re-assembles) every module in a large program. A batch file consists simply of the list of commands that you wish to execute and has the extension .BAT. To run a batch file, just type its name from within Batcher. If you are using the GEM Desktop you can just double-click on the batch file, if you have installed Batcher for the document type .BAT.

When Batcher is executed without a command line, it looks for a batch file called AUTOEXEC.BAT and executes the commands contained in this file. The most common use of this facility is to set up environment variables, and your preferences, but you can use it for any purpose.

Batch files may have parameters. Within the batch file these are referenced as %1, %2 up to %9 for the last parameter. To include a % character in a batch file use %%.

For example if mv.bat contained

```
del %2
ren %1 %2
```

Then:

```
mv  mine.c new.c
```

would rename mine.c to new.c deleting any old version of new.c.

If the first two non-space characters on Batcher's command line consist of /C, Batcher treats the rest of its command line as a single command. This facility is provided so that you can use Lattice C system function to run Batcher commands from within your own programs. You need to set the COMSPEC environment variable so that the system function will know where to find Batcher.

Redirection

Batcher supports command line re-direction. To cause the output from a program that would normally be sent to the screen via GEMDOS to be sent to a file, add > (greater than) and the name of the file. e.g

```
dir *.c >mydir
```

will create a list of the C files in the current directory to the file mydir. You can also append to an existing file by using >>, thus:

```
dir *.h >>mydir
```

would add the .h files in the current directory to the end of the file, mydir. You can also re-direct input to come from a file using <.

Note that all redirection operators must be at the end of the command line and that due to problems with the operating system, some programs may behave strangely if their input/output is re-directed. Some of these anomalies are described in the **Volume III - Atari Library manual**.

WERCS
The Resource Editor

WERCS is an acronym for WIMP Environment Resource Construction Set and is pronounced *Works*. It allows you to create and edit resource files for use with GEM programs.

What is a Resource File?

A resource file is a special file (normally with the extension .RSC) that contains *resources*. A resource is actually a *tree* structure in memory which is used by the GEM AES to produce such things as:

- Menus
- Dialog boxes
- Icons
- Alert boxes
- Strings

A resource file contains such things to deliberately keep them separate from your program code. In addition, the X-Y co-ordinates of every item in a tree is stored in such a way as to produce the same visual layout, regardless of the screen resolution. This means one resource file can be used for all screen modes and by many different programs.

Using resource files is good practice because it encourages modularity and aids portability thus saving you time and energy in the long run.

A certain understanding of the way a resource file works is required in order to create and use such a file.

Each resource file contains one or more *trees*. A tree may be one of five different types: *Form, Menu, Free String, Alert* or *Free Image*. Forms and Menus are the most common; each of these, in turn, consists of individual *objects*, where each object has a distinct type, use, purpose and appearance.

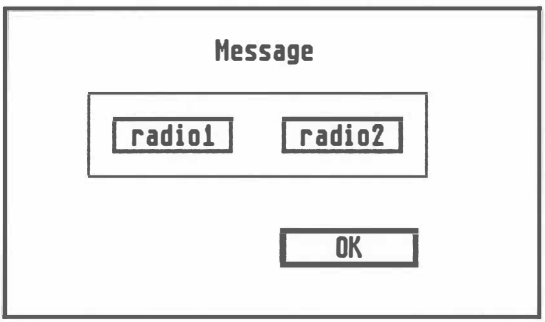
Whilst you are learning to use WERCS we recommend that you start off by using just Forms.

What is a Tree ?

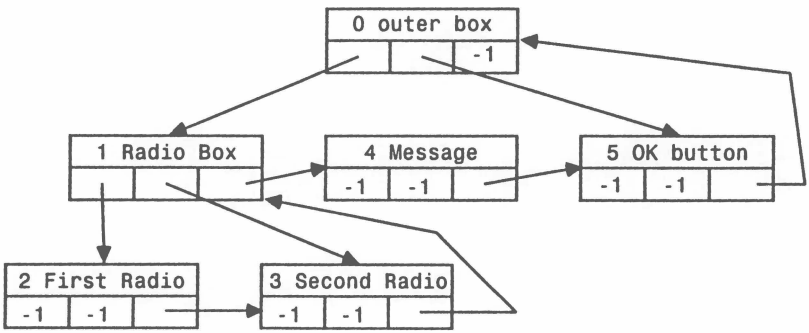
Forms (or Dialog Boxes) and Menus are GEM AES *object trees* and to understand resource files you need to understand the structure of object trees.

Many of the WERCS commands work on parts of object trees and we shall use tree terminology to describe them.

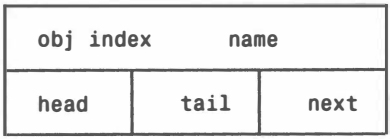
When it is loaded into memory an object tree is like an array of records, each record describing an object. The first object (with index 0) is called the *root* object. It is normally the outer box of a Dialog Box. Each object in the tree has eleven fields. Three of these fields, the *head*, *tail* and *next* fields, hold integer values that dictate to the AES the structure of the tree. Fortunately you do not normally need to access these directly, WERCS does it for you. As an example, say we have a Dialog Box like this:



The tree structure this represents can be shown as:



where the components of each box are:



Most of the terminology used to describe object trees is similar to that used in human family trees; of course objects only have one parent and most people don't think of themselves as ultimately descended from a root!

Object number 0 is called the *root* of the tree. Its *children* are Message, Radio Box and OK button. Radio box's *parent* is outer box; its children are First Radio and Second Radio; its *siblings* are Message and OK button. First radio and Second radio are *childless* and they are *grand children* of the root object, outer box.

Normally what is important with object trees is the tree structure, not the order that the items are in memory. The detail of how trees are stored in memory is described in **Appendix B - Resource Details**.

What is an Object?

There are thirteen types of object that you can have in object trees; most of them are some form of text or boxes or a combination of both. Different types have different memory requirements; in general the more flexibility the more bytes are used.

As well as the fields described above, associated with each object is its *position, size* and also some *flags* and *states*.

The position of an object is always given relative to its parent; normally you set the position and size of the object using the mouse - WERCS takes care of the calculations for you.

The flags and states are used for two purposes; first to change the appearance of an item; for example whether a box has an outline (Outlined), and also to give information to the AES; for example that clicking on a Button will cause control to be transferred back to your program (Exit). To start off with you need not be too concerned about flags and states as WERCS gives you sensible defaults.

The thirteen types of objects are as follows:

- Box — A straightforward box - can have a fill pattern and a border.
- IBox — An 'invisible' Box; only truly invisible if it has no border.
- String — A straightforward string of characters.
- Button — Like a String but with a box round it; normally used for Dialog Box buttons.
- Text — Like a String but with more formatting possibilities: colour, size and justification.
- BoxText — Like Text but with a surrounding box as well.
- BoxChar — A single character in a box. The most memory efficient way to have a single character in a filled or coloured box.
- Title — A special form of String only used in Menus.
- FText — This is like Text but can be used for editable text so that you can type in characters, numbers etc. The programming interface isn't easy but we show you how to do it in the example program.
- FBoxText — Like FText but with a box around it.
- Image — A simple bit-mapped graphic image.
- Icon — Like an Image, but with a mask so that it changes sensibly when selected and also has a character and string associated with it. Originally invented for the desktop's disk icons.
- ProgDef — A programmer-defined object with its own drawing routine. We recommend that you don't try these out until you've exhausted the possibilities of the pre-defined objects.

Header Files

In order for a program to use a resource file, the programmer must be given a method of referring to each tree and object; WERCS helps you by creating a *header file*, as well as the actual resource file. As you create a resource file you can give names to both trees and objects, so that you can refer to these names within your program. The header file contains constants which translate these names into integer values. The header file is then #Included as normal into your program.

If the compiled version of the header file is out of step with the resource file, strange things will happen; this varies from slight mis-behaviour to total system crashes.

Quick Tour

Running WERCS

To run WERCS, simply double-click on the WERCS.PRG icon. WERCS also needs its .RSC and .LNG files in order to run.

There now follows a whistle-stop tour of WERCS introducing the editing facilities available. A more detailed reference section is to be found in the next section.

Low Resolution

WERCS runs in all screen modes, for maximum flexibility. When running in low-resolution, the title of each menu is reduced to the first two characters only. However, the full menu title is shown at the top of the menu box, once it has been pulled down.

Creating a New Resource File

Having loaded and executed, WERCS will display a *tree window*, labelled Untitled. A tree window displays all the trees within the file and initially this is blank since you are starting with an empty file.

When creating a new resource file it is best to select the programming language for which you require the header file before you enter any names. This can be done by selecting Language from the File menu. You can also choose whether your names will be upper-, lower- or mixed-case. Selecting the language before you start ensures you don't make any naming errors while building the file. Naturally C is the default language with this version of WERCS.

Creating a New Tree

To create a new tree you simply select a suitable type of tree from the Tree menu - this stage is known as *tree-level editing*. An icon representing this type of tree will appear in the tree window, together with a dialog box. The main point of interest for the time being is the name that you wish to call the tree - when you are happy with its name press Return and you will then be at the *object-editing* level.

Creating Objects

WERCS will now display a window showing the new tree, allowing you to add and edit new objects. To add an object, select the object type from the Object menu. The mouse will change into a representation of that object, then you should click where you require the object to be placed. To name this object, double-click on it, to move it simply drag it, or to re-size it click on its lower right-hand corner.

When you are satisfied with the objects in this tree, clicking on the Close box will return you to the main tree window. If you don't like anything you have done, the last session may be aborted by selecting Abandon Edit from the Edit menu.

When the tree window is visible, an existing tree may be edited by double-clicking on its icon. Certain attributes, such as the name, may be changed by single-clicking to produce a dialog box.

When you are happy with your resource file, ensure the correct language choice has been made then Save As the file. This will create the .RSC file containing the actual resources, a .HRD file containing your names for each item, and a header file.

Using WERCS

General

Most of the editing actions inside WERCS are obtained from menus or via the corresponding keyboard short-cut. Keyboard shortcuts are shown in each menu with a ⌘ symbol denoting the Alt key, and ^ denoting the Ctrl key. Menus that are inapplicable at a particular time are disabled. Owing to a bug in the original version (1.0) of the operating system, the titles are not disabled when using these ROMs, although naturally these commands will have no effect. There is a summary of the keyboard short-cuts at the end of this chapter.

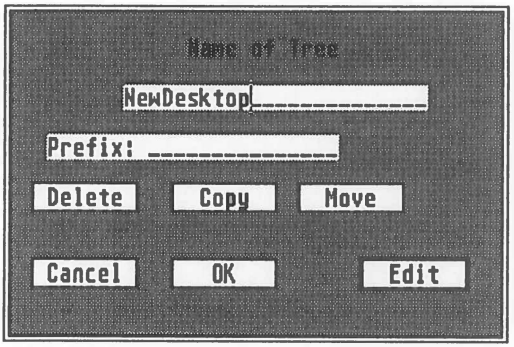
Introduction to Creating and Editing Trees

There are two main levels when running WERCS. The *Tree Level* is used for manipulating which trees are in your file and the *Object Level* for the items within those trees.

When you open a file (or use New) a window containing the trees in the file is shown, known as the *tree window*. You can add a new tree to the file by clicking on one of the items on the Tree menu: Form, Alert, Free String, Menu and Free Image. Whilst you are learning to use WERCS and if you are not familiar with GEM it is best to just use Forms. Forms account for the vast majority of trees in any case.

Note that a Form is also known as a Dialog Box; we use the terms inter-changeably. Generally we use Form in the context of editing and in the programming section we refer to Dialog Boxes.

After clicking on Form from the Tree menu you will be presented with a dialog box like this:



You can now enter the name of the tree. The defaults for these are MENU1, MENU2, FORM1, FORM2 etc.

Pressing the Return key or clicking on the Edit button will then let you edit the objects within the tree. The other buttons and fields are described in detail later.

To edit an existing tree, such as a Menu or a Form, double-click on the appropriate tree icon. You will then be taken straight to the Object Level.

Changing Objects

Once you have clicked on Edit from the Name of Tree box you are ready to add objects to the tree. To add a new item to a tree, click on the required item type from the Object menu. The mouse will change to an outline representation of the item that you are adding. Release the mouse button and move the mouse to where you would like the new item, then press the mouse button. If you decide you do not want to add this item after all, click on the Cancel item from the Edit menu.

When you have finished editing a form click on the Close box; this will return you to Tree Level mode.

Selecting objects

To change the attributes of any object, single-click on it. It will then be *selected* (highlighted) and you can use most of the menus to change its attributes, the border for example. The exceptions to this are the text items, Images and Icons; to edit these, double-click on an object.

When the object is selected the GEM *selected* bit is used to show this. This means that if you click on a box it will appear black. In particular if you click on the outer box it will all go black. If you didn't mean to click there you can either:

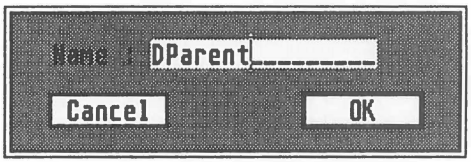
- click on the menu item Cancel from the Edit menu,
- click outside the box, or
- click on the item that you meant to select.

If you wish to edit the *parent* of the current object, single-click with the ALT key held down - the parent of the object will then be selected. If you already have an item selected and ALT-click again, its parent will be selected. This may be repeated any number of times until the whole tree is selected. To bring up the Text Box of an object's parent double-click whilst holding down ALT.

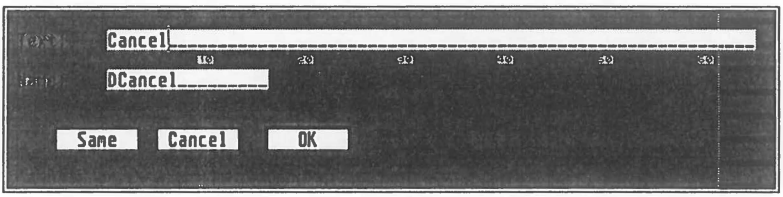
Item Names and Text

To change the Text or Name (remember: Text is the displayed message; Name is what your program will know the item as) of an object, double-click on that item - this will present a Text dialog box. This varies depending on the item.

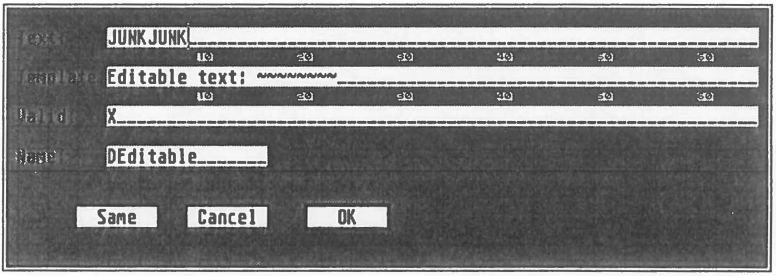
For example a Box only has a Name and presents a dialog box like this:



Buttons have one Text field and so have this type of box:



Whereas FText and FBoxText items have the appropriate TEDINFO fields as well:



To make the Name the same as the Text, click on the Same button - this is like clicking on OK except that the object will be given a name based on the text of that object and the Prefix for this tree, if any. This can save a great amount of tedious object naming. For editable text the name is taken from the Template thus giving the same name as your prompt.

Since underline characters are used by WERCS in a special way then, when editing text fields, you should enter underline characters (_) as tildes (~) and vice versa. If we didn't do this it would be impossible to see how many underlines you have in your Template strings.

To enter control characters into strings enter \\ (two back slashes) followed by the ASCII symbol corresponding to the control character. For example the ALT key symbol is entered as \\7. Similarly the copyright symbol (©) is \\189. Don't try and enter a null character \\0 as the AES treats this as the terminator of the string.

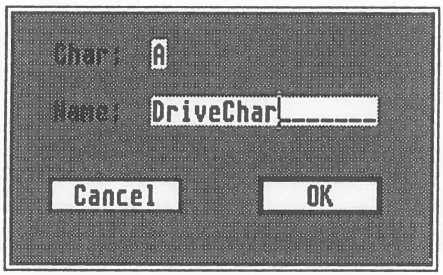
In the unlikely event that you need to enter two consecutive back-slashes type three instead; for three back-slashes type four and so on.

When using formatted text (FText or FBoxText) you should ensure that the Text field has the same number of characters as the Template field has ~s (stored in the file as underlines). If you are using different Valid characters then you should have the same number of characters in the Valid field as there are ~s in the Template field.

If you are using the same character throughout the Valid string you can enter just one as in the example above. We have not seen this facility officially documented but it works with all known versions of the operating system at the time of writing.

The other attributes of TEDINFOs (such as Large/Small characters) are set using the Text menu.

BoxChars (single characters surrounded by boxes) have their own dialog box, thus:



Moving and Sizing Objects

To change the size of an item, place the mouse near its bottom right-hand corner and drag to the required size. By *near its bottom corner* we normally mean within *one character cell* of the bottom corner but inside the object. If the object is less than a character high then you should click within the bottom half-character of the box. Similarly if it is less than one character wide you need to click within a half-character of the right.

Note that the border of an object is often outside the object itself as is any outline or shadow of a box. This area is not considered part of the object by the objc_find call. This means that you will not be able to select or drag an object by clicking in its border, outline or shadow. Also, any program you write to handle such objects must bear this in mind.

To move an object within a tree, drag from somewhere other than the bottom right-hand corner. If the object has any children, they will move with the object.

To move or size any parent or siblings of an object, first select the required objects using ALT-clicking as described under **Selecting Objects** above and then drag as if you were moving a single item.

If you want a quick copy of an object or objects, select and then Shift-drag. Generally it is best to Shift-click with the mouse near the top left corner of the object as this is where the object will appear. This will let you move a new copy of the object leaving the old one where it was, so you can drag the new one to its new position. This is particularly useful if you have a set of similar objects with the same flags and attributes set (say disabled, right justified small Text). Set up the first one and then Shift-drag to create the rest.

If you using Shift-clicking to move an object which has children, you will get copies of the children too. When you let go of the mouse you will be asked whether you wish to delete the children too. Indicate that you do wish to delete the children; otherwise you will get an extra set of children, starting where you originally clicked.

If you move an object outside its parent then you will be asked for confirmation of this (unless you are in Expert Mode).

If you move an object so that it would completely cover another object or objects then you will again be asked if you wish to adopt these objects as children of the object that you have moved. If the new position of the object will partially cover another then you will be given an error message.

After you have moved or sized an object it may be snapped to the nearest character or half-character boundary, if you have used the Auto Snap or Half Character Snap commands.

You may also change the position and size of an object using the Extras command from the Flags menu.

Editing Images

Double-clicking on Images will bring up the Icon Editor which will give a screen display similar that below:

The largest and main part of the display is used for editing the Image a pixel at a time. Beneath this is an Actual Size representation of the Image, as it will appear in your form and to the right are various buttons that you may click on.

To change an individual pixel, just click in the appropriate place on the screen; if it was black it will become white and vice-versa. To make a number of pixels the same colour click and drag; note that the actual size display will only the updated when you release the mouse button.

At the top of the button area are the buttons for changing the *height* of the Image together with the current height (28 in the example above). To increase the height click on H+ and to decrease it click on H-. Both of these work one pixel at a time and will repeat if you hold the mouse button down. The size of the main display changes to ensure that it is as large as possible whilst still displaying the Image at actual size beneath it.

If you decrease the height by too much, increase the height again and the newly displayed area will be the same as it was before you made the Image display smaller. The maximum height of Image that you can edit is 128 pixels. If you attempt to edit a larger image, it will be truncated to 128 pixels high.

To change the *width* of the Image click on the W+ and W- buttons; the current width is displayed to the right of these buttons. GEM restricts the size of Images to multiples of 16 pixels (so that it can draw them on the screen quickly) so these buttons change the width 16 pixels at a time. The width of the main display and the button will change to give as large a main area as possible. These buttons do not repeat if you hold them down. As with height changes, do not worry if you make the width too small by mistake, just click on W+ and the area that you have just deleted will re-appear.

The maximum width of an Image that can be edited is 128 pixels. Again, editing an image that is more than 128 pixels wide will cause it to be truncated.

The arrow buttons scroll the main display in the appropriate direction. Scrolling upwards and to the left loses the pixels that are removed from the Image. The pixels that are lost when scrolling to the right or downwards can be retrieved by scrolling to the left and upwards, assuming that the maximum size of 128x128 is not reached.

The Clear button will clear the entire Image to white; unless you are in Expert Mode you will be prompted to check that this is what you want.

The Fill button will fill the entire Image to black; as with Clear you will normally be prompted for confirmation.

VFlip and HFlip reflect the Image in a vertical/horizontal line through the middle of the Image. The best way to understand this is to try it. Clicking on VFlip (or HFlip) twice is like doing nothing at all.

Line is used to draw a line of black pixels. The mouse cursor will change to a +; click where you would like the line to start and then on where you would like the line to finish.

Cancel is used to cancel all the changes that you have made since entering the Image Editor; if you are not in Expert Mode you will be prompted to ensure that this is what you require.

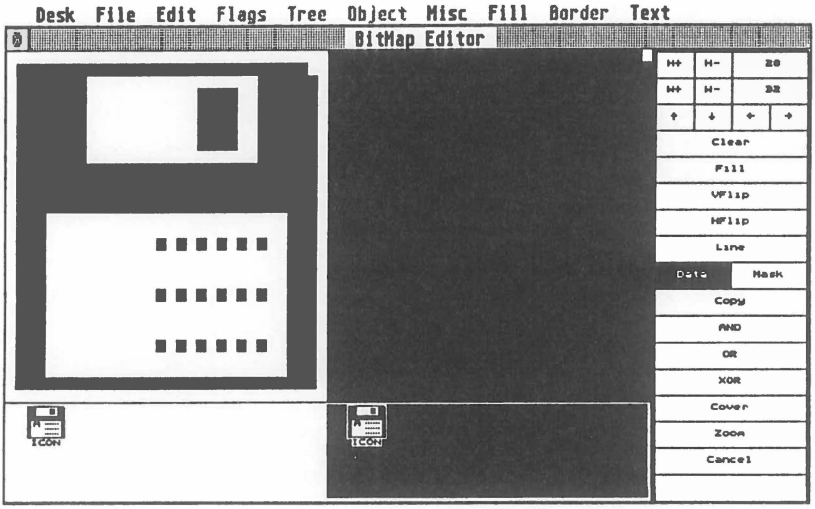
To name an Image, double-click in the Actual Size area; the usual name box will then be displayed.

The Text menu can be used to set the foreground colour of the Image.

The normal way to exit from the Image Editor is via the Close box although you can also use the commands on the File and Misc menus.

Editing Icons

The display when editing an Icon is like this:



Editing an Icon is like editing an Image except that the main display consists of the Data of the Icon on the left and the Mask of the Icon on the right. There are also some extra buttons in the Icon Area, and the Icon's string and character fields can also be accessed.

There are two Actual Size displays; the one on the left normally shows the icon not selected; that on the right shows it selected. If however you set the Selected bit using the Flags menu from the Object Level window then these will be the other way round. The Icon on the left is always as it will appear in the file.

The extra buttons for Icons are used as follows:

Data and Mask are a pair of radio buttons; if Data is selected then the Data bit map is used as the source for the commands below and also as the current bitmap for Clear, Fill, VFlip and HFlip; WERCS will remind you which bitmap you are destroying unless you are in Expert Mode. The rest of the commands are described assuming that Data has been selected; to perform the action the other way click on Mask to select it first.

Copy, AND, OR, and XOR perform the appropriate logical operation; so that Copy will make the Mask the same as the Data; AND will set only the bits in the mask that are already set in both the Mask and the Data; OR will set bits that are set in either or both and XOR will set those bits that are different in the two bitmaps.

Cover will copy the mask and surround the bits that are already set with extra bits. The source bitmap should not have any pixels set around each edge as these will be cleared in the source so that it can be covered correctly. This is useful for producing the first attempt at an Icon's mask from its Data bitmap; this is another command that is best understood by experiment. As usual you will be warned about the area that will be destroyed before proceeding if you are not in Expert Mode. If you delete something unintentionally you can always use Cancel to revert to the Icon before you entered the Icon editor.

Zoom causes the current selected bitmap, Data or Mask, to take up the whole of the main display; this is intended for editing large Icons where each pixel is very small in the main display. Click on Zoom again to return to the normal display.

Visually a finished Icon has three components; the Bitmap part, the String (ICON in the example above) and the Character (A in the example above). Every object of type Icon has an overall size just like any other GEM object; this is normally bigger than the Bitmap itself. The three components may each be positioned independently relative to the top left corner of the object. In the example, the Bitmap is to the left of the box and the Text near the bottom in the middle. Strangely the single Character's position is actually relative to the bitmap not the main object. Also GEM will draw the Bitmap and String even if they are outside the object's box.

To edit the text of the Icon's String, double-click on the text in the Actual Size display; this will bring up the usual text name box so that you can enter the String and also set the Name of the Icon object. The Icon Text may also be moved in the normal manner. Editing the Icon's Character works in a similar way and you may also move the Bitmap itself within the nominal box represented by co-ordinates of the object. The object's co-ordinates are changed as usual in the Object Level window.

Frequently Icons do not need either or both of the String and Character attributes; you can just set these to be blank. So that you can edit such text, on entry to the Icon Editor, blank strings are represented as _____ and blank characters as _. As a result of this and the fact that the actual display for icons is simulated (so that WERCS knows accurately where the components are) the Actual Display is not quite the same as the GEM display in the main Object Level window or when the Icon is displayed by your program.

We will now describe the menus in detail.

File Menu

The File menu is used to manipulate which file you are editing. Initially you are editing a blank file, shown within a window labelled Untitled.

New

To starting editing a new empty file, click on the New item from the File menu.

Loading

To load an existing resource file click on Load and select the appropriate file from the File Selector. An alert box saying .HRD file not found will appear if this file is missing - you will still be able to edit your file although any names previously attached to trees or objects will be lost.

Another way of loading a file is to set up the GEM Desktop to load WERCS when you click on .RSC or .HRD files, using Install Application. Similarly you may invoke WERCS from a CLI (such as Batcher) in which case the file extension is not required.

If you wish to load a resource file created with another resource editor you may like to convert the other editor's equivalent of the HRD file into a true .HRD file, in order to preserve the names of your items, using the conversion utility WCONVERT.

Importing Images

Images and Icons converted using the WIMAGE utility can be imported using the Import Image item on the File menu. You will be presented with the File Selector to enter the file to import. This will copy the object to the Clipboard, subsequently selecting Paste from the Edit menu will place it in your file.

This command actually copies the second object from the first tree in the file to the Clipboard.

Saving

Clicking on Save As will present you with the standard file selector and you can choose a filename for the current file. Clicking Save saves the file, without pause, under its original name - if the file was Untitled then Save will do a Save As.

The filename you enter into the File Selector for Save As need not have any extension - suitable extensions will be added by WERCS. In addition to a .RSC file, an .HRD (for HiSoft Resource Definition) file is also saved. This is a special file which contains such details as the names you have selected for the contents of that file and which other language files are to be created. This method ensures that, once you decide a particular resource file is to be used for C, for example, WERCS will know each time you edit it. The .HRD file format is described in detail in **Appendix B**.

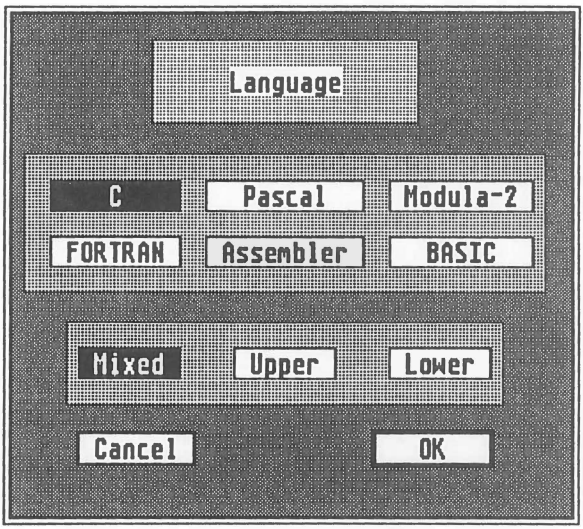
Save Prefs

The default values for various options such as the language that you are using and the character snap for new files when WERCS is loaded are read in from a file called WERCS.INF. This is searched for on the standard GEM path.

To change the defaults use the Save Prefs item on the File menu.

Language

To change the files that are created when you use Save, click on the Language item on the File menu. This will present you with the following dialog box:



You can select the language used for the header file when you next Save the resource file. It also allows you to select the names to be in lower-, upper- or mixed-case. Details of supported languages and file extensions may be found in **Appendix B**.

Quit

To leave WERCS, click on Quit. If you have changed the file you are editing you will be given the opportunity to save or lose your modifications. You can also use the Close box on the tree window to achieve the same effect.

Flags Menu

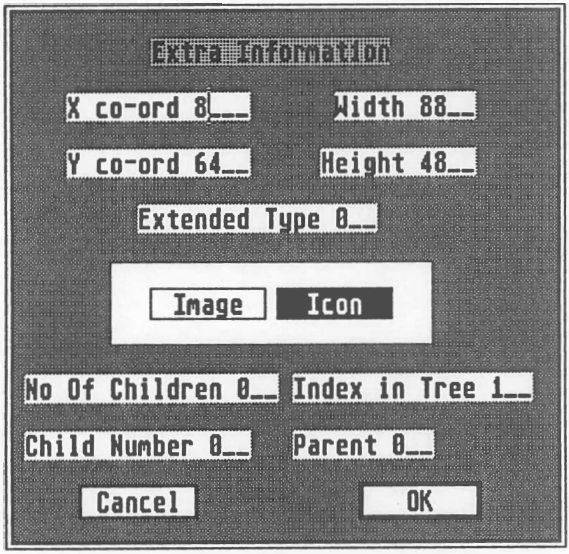
This menu contains the various attributes that are part of the ob_state and ob_flags fields of object tree items. A selected item is shown ticked. The corresponding standard GEM names for the fields are as follows :

Selectable	SELECTABLE
Default	DEFAULT
Exit	EXIT
Editable	EDITABLE
Radio Button	RBUTTON
Touch Exit	TOUCHEXIT
Hide	HIDETREE
Selected	SELECTED
Crossed	CROSSED
Checked	CHECKED
Disabled	DISABLED
Outlined	OUTLINED
Shadowed	SHADOWED

UnHide Children

The item Unhide children will clear the HIDETREE bit for any immediate children of the selected object so that they become visible and you may then select them once more.

This displays a dialog box similar to that shown below which allows direct access to the object's internal structure and thus care should be taken when using this command.



The X, Y, Width and Height items are relative to the object's parent in pixels. These may be modified; use this with care as you can easily make objects move outside their parents.

The Extended Type is the most significant byte of the object word. This is ignored by the AES but may be used for your own purposes.

No Of Children is the number of first generation children that an object has. It does not include 'grand-children'.

Index in tree is the object number relative to the root object of the tree.

Child number is 0 for the first child of its parent, 1 for the second and so on. This field may be changed, in which case the objects between the old and new positions will change position in the tree. The easiest way to place the children of a particular parent in a particular order is to select the first child and make this child 0 then select the second child and make this child 1, etc.

Objects may also be re-ordered using Sort from the Misc menu.

Parent gives the Index in tree number of the object's parent.

The buttons in the box (Image and Icon in the above example) tell you what type the selected object is and let you change the object's type. Image may be changed to Icon, Box to IBox, String to Button, and Text, FText, BoxText and FBoxText interchanged.

Changing Icons into Images loses the mask and string items of the Icon so you are prompted for confirmation unless you are in Expert Mode.

Fill Menu

The Fill menu lets you change the fill pattern of the object, the colour of the fill and whether it is *opaque* or *transparent*. Opaque means that text will be displayed with a white background whereas transparent means that the fill pattern and fill will show 'behind' the text.

The fill pattern and colour are only applicable to Box, BoxChar, BoxText and FBoxText objects. The transparent/opaque setting is only applicable to BoxText, Text, FBoxText and FText objects.

The Fill menu is also used to set the background colours of icons.

Border Menu

Lets you change the colour and size of the border for an object. Clicking on the Size item brings up a box as below:

The Border Size is specified in pixels as appropriate. A negative size means the border is drawn inside the box; a positive number means it is drawn outside. This is only normally useful with Box, BoxChar, BoxText, FBoxText and IBox objects.

If set for FText or Text objects, the border affects the size of the box that is drawn when the object is selected thus increasing or decreasing the visual size of the object.

Text Menu

The Text menu lets you change the justification, colour, and size of the text object types: BoxChar, BoxText, FBoxText, Text and FText. The actual text is changed by double-clicking on the object.

Clipboard

The clipboard is a special area of memory which can contain trees or objects. It is ideal for moving or copying items between different areas of a resource file, or between different resource files. All clipboard commands can be found on the Edit menu.

Cut

Cut will copy the currently-selected object to the clipboard with its children and removes the current object from the tree. If the object has children you will be asked if you wish to delete them as well. If you choose not to delete the children they will become the children of the deleted object's parent. The object may then be pasted somewhere else.

Paste

Changes the mouse form to a pointing finger and waits for you to left-click. This will place a copy of the object at that position. To cancel this, click on Cancel on the Edit menu.

Copy

Copy copies the current selection to the clipboard and leaves it in place.

Cancel

Cancel is used to cancel the selection of a menu when the mouse has changed to a non-pointer form. For example, if you click on String from the Object menu and decide that you do not want a new string after all, click on Cancel.

From within the Image/Icon editor, Cancel will cancel all the changes you have made since you entered the Image/Icon editor. You are prompted with a dialog box first to ensure that this is really what you wish to do.

Abandon Edit

This allows you to abort the object-level editing that you are currently performing. All changes made since you chose to edit the current tree will be lost. It's ideal if you have made a major mistake in editing a particular tree.

Delete

Delete works like Cut but leaves the contents of the clipboard intact.

To copy just some of the objects of a parent, create a new Box using the Object menu and make this cover the objects you wish to copy. Then select Copy and use Delete (*not* Cut) to delete your temporary Box, but not its children. Now use Paste to place the copy of the objects where you need them and then Delete to remove the outer Box again. This sounds more complicated than it is in practice.

Misc Menu

Auto Size

With Auto Size enabled, every time you change the text of an object the size of the object's box will change to just surround it; thus if you make the text of a Button longer it will make the Button bigger; shortening the string will make the box smaller. If you switch Auto Size off, the Button would stay the same size and the new text would not necessarily fit in the existing box.

Auto Naming

If Auto Naming is enabled (shown by a tick) then objects are automatically given a Name as if the Same button in the Text dialog box had been clicked. The Name is based on the tree's prefix, if any, and the Text of the item.

Auto Snap

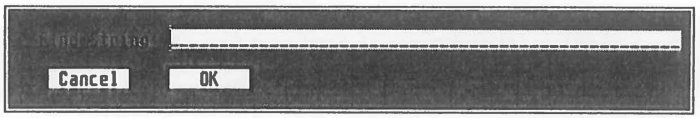
If this item is selected from the Misc menu then every item that you move or size will be *snapped* to the nearest character boundary. This is useful to make sure that items line up and will appear the same in different screen resolutions.

Half Char Snap

If this item is selected from the Misc menu then objects will snap to the nearest half-character boundary, in a similar way to character snap. However if you are designing a resource file to run in more than one resolution then objects will not necessarily come out the same, as half a character in one resolution may be either a whole character or a quarter of a character in another resolution.

Find Text

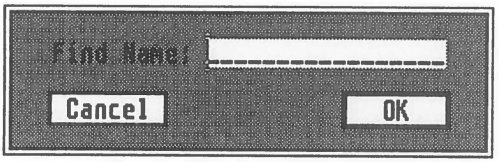
This enables you to find occurrences of a particular string within the text fields of the objects. You are presented with a dialog box, as below,



For example, if you have a number of menus and cannot remember which menu contained the item Stop you could use this command. The appropriate tree is opened and the object containing the string is selected.

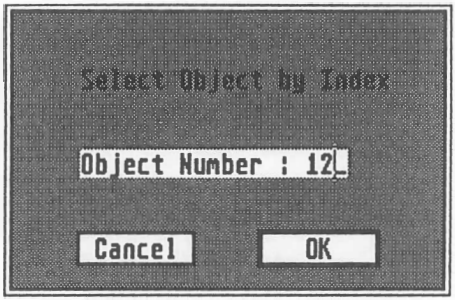
Find Name

This item searches for a particular named object within the file, opens the appropriate tree and selects the object. The box presented looks like this:



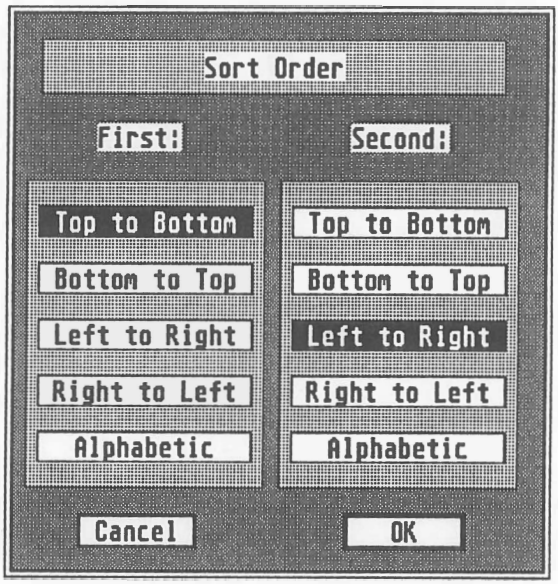
Number Select

This allows you to select an object given its *object number* in the current tree; this can be useful if you have an object outside its parent.



Sort

This enables you to sort the children of a particular object according to various possible criteria that are selected from the dialog box:



Top to Bottom and Bottom to Top will sort the objects according to their y position on the screen whilst Left to Right and Right to Left will sort them according to their x position on the screen. There are two priorities for the sort, First and Second. Note that the sort *does not* affect the objects' positions on the screen, it affects their order *in memory* and *within* the tree.

The default is First, Top to Bottom and Second, Left to Right. So, say we have 6 objects (names objA to objF in any order in the tree and in memory) with screen representations as follows:

objD objA objE

objB objF objC

and then we sort using the default options. The objects will be sorted so that their order in memory and in the tree is objD, objA, objE, objB, objF, objC. Note that the sort will *not* affect the screen representations.

Alphabetic means that the *strings* of the objects are compared rather than their screen positions. Sorting alphabetically does *not* mean that the objects will change position on screen only that their position in the object tree and in memory may change.

Remember to select the parent of the objects that you wish to sort before you click on Sort. You can use Alt-clicking to select the parent of a given object.

Test

Test lets you test out a Form, Menu or Alert Box. In order to test a Form it must have an object (such as a Button) with the EXIT and SELECTABLE flags set, or alternatively with the TOUCHEXIT flag set. If it does not then you are given an error message.

When you click on an Exit Button (or click on an item if testing a Menu) then you are told which item you have selected and its name, if any. You can choose whether to continue testing the tree, or return to WERCS. If you double-click on a TOUCHEXIT item then the value displayed will not include the top bit, as returned by form_do.

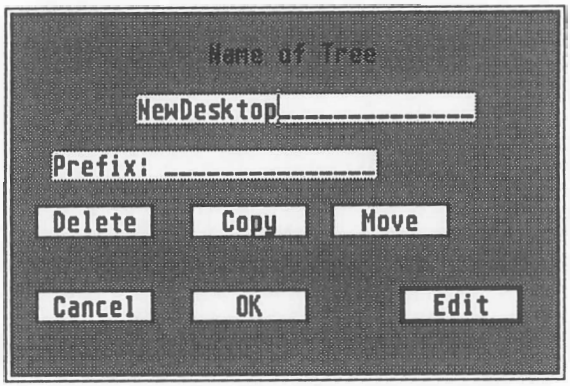
Expert level

If this is enabled (shown by a tick) then all warnings to do with tree re-organising are suppressed: for example, when an object is given a new parent, or commands that effect the entire data or mask bitmaps in the icon editor. This also includes the warning about losing information when changing an Icon to an Image using Extras and when using the Abandon Edit command.

Tree Level Editing

Forms

Forms are the most common type of tree in resource files; they are normally used for dialog boxes, but can also be used for replacement desktops, to change the pattern of the background in a GEM program or to add icons to it. When you create a new Form or single-click on an existing one in the file window, you are presented with a dialog box like the one below:



The name of the tree may be changed. WERCS will check to ensure that it is a valid name according to the current selection of Language, with the correct case, and that it is not a duplicate of a current name. If the other item with this duplicate name is an object you can use the Find Name command to select it and then change its name.

Remember that in Mixed case Ok and OK are different names but if you have selected Lower or Upper case then they are not.

Pressing the Return key or clicking on the Edit button will then let you edit the objects within the tree.

Cancel will not add a new tree to the file and will disregard any changes to the tree name that you have made.

To add more than one tree without editing them immediately, click on the OK button - this lets you set up a number of trees without entering the objects.

To re-order the trees in a file, click on the Move button. This will change the mouse form to a pointing finger; you should then click on the tree that you wish to place immediately *after* the current form. Owing to the structure of resource files, when the file is reloaded, the Menus and Forms will be first, followed by the Free Strings and Alerts, followed by the Free Images.

To delete or copy an entire tree, click on the appropriate button. To paste a tree from the clipboard into your file, click on Paste from the Edit menu.

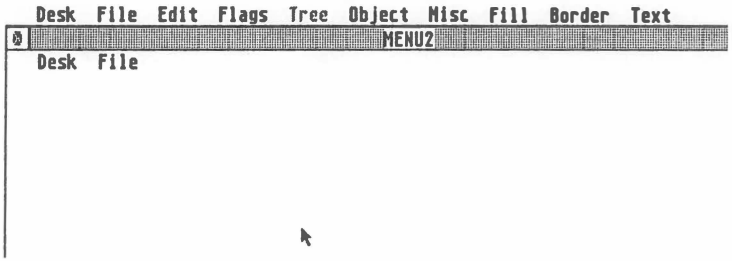
This box also lets you set up the Prefix for this particular tree. This is used to provide the start of the names of objects if you use Auto Naming.

To edit an existing tree, double-click on the appropriate tree icon. You will then be taken straight to the Object Level.

Menus

Menus are a very special type of Form which must conform to a number of unpublished rules, otherwise GEM will behave strangely. Fortunately, when using WERCS, you don't have to worry about these rules as WERCS will cope with them for you.

When you ask for a new menu you will see a screen similar to that below:



Normally Menus consist of Titles (which are displayed along the top of the screen) and Strings (which are displayed in the pull-down menus themselves). To add a new Title, click on Title from the Object menu and then click in the menu bar where you would like it. The other Titles (and their menus) will be moved if required.

To add items to a given Title, first click on the Title itself; this will cause the appropriate Menu to appear, for example:

You can then insert objects in the usual manner, normally Strings, and objects below the new object will be moved down. You should ensure that the mouse pointer is at the left edge of the box when inserting objects; otherwise you will leave a 'hole' to the left of it.

You can use types other than Strings in Menus if you wish; we used WERCS to produce its own resource file, for example. You can also change the flags and states of items just as if you were editing a Form. For those objects that have them, the items on the Fill, Border and Text menus can be used.

If you want your menu to work in more than one resolution don't use Icons or Images or you will find that there will either be gaps between them or they will overlap. This is because the width and height of these objects are a different number of character cells in different resolutions. This can be avoided by adjusting the object once it is loaded so that there are no gaps between icons.

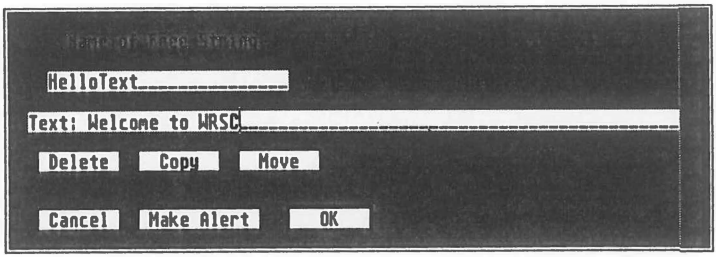
The boxes surrounding menus must not be take up more than one quarter of the screen, otherwise the system may crash. Be especially careful, when designing menus for use in Low Resolution.

The Tree Name box can be used in the same way as for Forms.

Free Strings

A Free String is a string of characters that is not connected with any particular tree. They can be used to facilitate foreign-language versions of software, for example.

The Name and Text of the Free String can be modified in the same way as any other type of string in WERCS so that you can use \\ to enter control and graphics characters for example.



To edit an existing Free String, it is only necessary to single- or double-click to bring up the box shown above.

The Delete, Copy, Move and OK buttons work in the same way as with Forms, detailed earlier.

Make Alert can be used to turn a Free String into an Alert. You should ensure that the String conforms to the rules for Alerts, as described below.

Alerts

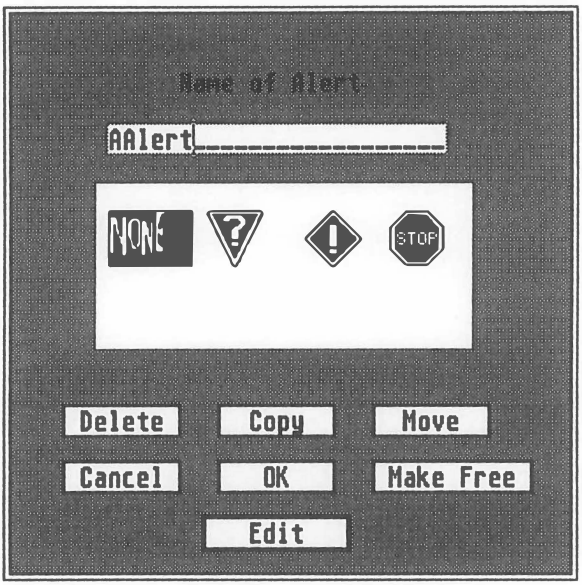
Alert Boxes are actually stored as Free Strings (see above) but are passed to the AES form_alert call to display an alert box.

There are two types of restrictions as to the contents of Alerts; the first type is those restrictions documented by Atari (to keep down the amount of memory used by the AES when displaying them) and the second type of restriction is caused by bugs in the first release of the operating system ROMs.

As officially documented, each line in an alert box must be no more than 30 characters and there is a maximum of 5 lines. Each Button must be no more than 20 characters each and there is a maximum of three Buttons. Strings and Buttons may not contain] or | characters. ROMs prior to 1.2 do not check for the infringement of these rules and failing to adhere to them will corrupt certain areas of the AES workspace! WERCS rigidly enforces these rules when converting the tree representation back to a string.

The release 1.0 of the ST ROMs contained various bugs, including long buttons/short text problems. Before releasing a commercial program ensure you have checked all your Alerts on a 1.0 ROM machine. Subsequent ROM releases (1.2 a.k.a. Blitter TOS, 1.4 a.k.a. Rainbow TOS and 1.6 a.k.a. STE TOS) have these problems corrected. These difficulties with the early ROMs have not been documented and so WERCS cannot check reliably for them. If you have later ROMs you have the flexibility to use some Alerts that it is not possible to use with the earlier ROMs.

The dialog box that you are presented with, when you single-click on an Alert Box in the file window, looks like this:



Which icon will appear in the Alert Box is controlled by clicking on the appropriate icon in the tree display as above.

The Delete, Copy, Cancel, OK and Move buttons work in a similar way to those on the Form Tree Name dialog box.

Clicking on Make String turns this item into a Free String rather than an Alert.

Clicking on Edit (or double-clicking on the the icon from the tree level display) will open an Object Level window that looks similar to that opened when you are editing a Form.

You should only add Strings and Buttons to the Form and these will be re-positioned automatically by WERCS. You can edit the Text in the normal way and also delete, copy and paste objects. Modifying the flags and states of the parts of an object will not affect the final Alert Box.

Re-ordering the Buttons in Alert Box is achieved by dragging a Button. If you drag button A onto Button B then the Buttons will be re-arranged so that Button A is immediately before Button B. This is similar to the moving of Titles in Menus. If it sounds complicated, experiment and you should soon get the hang of it.

Alerts are represented as strings of the format shown below:

```
[icon][line1|line2 ... |linen][button1|button2....]
```

where Icon is one of:

0 No icon
1 Question Mark icon
2 Exclamation Mark icon
3 Stop icon

line1, line2 etc. are the various message lines and button1 are the various Buttons. To check that you understand this, create an Alert and then change it to a Free String and, assuming that it is small enough to fit on the screen, you will be able to inspect it.

Free Images

A Free Image is an Image-type object that is not connected with any particular tree. When you use rsrc_gaddr you get the address of a BITBLK rather than an object.

The Tree Name dialog box for Free Images works in the same way as that for Forms except that clicking on Edit takes you straight to the Image Editor as for Image objects.

Keyboard Shortcut Summary

The following table gives the keyboard shortcuts when the Alt, Ctrl or Shift keys are held down.

Key	Alt	Ctrl	Shift
A	Abandon Edit	Border Size	Alert
B		Shadowed	Box
C	Copy	Crossed	BoxChar
D		Default	Form
E	Extras	Editable	Button
F	Find Text		FText
G	Find Name	Disabled	FBoxText
H	Selected Number	Delete	
I	Import Image	Small text	IBox
J		Large Text	Icon
K	Expert	Hide	Image
L	Load	Left	Free Image
M		Centre	Menu
N	New	Right	
O	Sort	Outlined	
P	Language	Opaque	ProgDef
Q	Quit	Transparent	
R	Save As	Radio Button	Free String
S	Save	Selectable	String
T	Test	Touch Exit	Text
U	Auto Naming	Un Hide	BoxText
V	Paste	Selected	Title
W	Auto Size		
X	Cut	Exit	
Y	Char Snap	Checked	
Z	Half Snap		

Note also that the Backspace key is used to delete objects, whilst the Undo key cancels an operation.

There is a rationale behind the choice of keyboard shortcuts to help you remember them; the Alt keys refer to commands on the File, Edit and Misc menus, Ctrl for the Flags, Fill, Border and Text menus and Shift for the Object and Tree menus. We have attempted to make shortcuts use the initial letter of the item as far as possible; the exceptions to this are the standard clipboard shortcuts and the following:

P Programming Language,

O Order (Sort),

^G Greyed (Disabled),

^B Border (Shadowed).

MonST2C
The Debugger

Introduction

MonST was originally designed as a low level debugger for debugging assembly language programs but we, and many other people, have found it useful in debugging C programs. The version of MonST that we supply with Lattice C, MonST2C, has been enhanced so that it 'knows' about where, in memory your program lines start and automatically loads your program and source code for you.

Together with the facilities of the original MonST, such as function labels, a full integer expression evaluator and named access to external variables, MonST2C gives you many of the features of a high level symbolic debugger. It does not give you access to local variables, floating point numbers, structures etc. As it was originally designed as a low level debugger some knowledge of assembly language is useful when using MonST2C.

As MonST2C uses its own screen memory, the display of your program is not destroyed when you single-step or breakpoint, making it particularly useful for graphical-output programs such as GEM applications. It also uses its own screen drivers so it is possible to single-step into the operating system screen routines such as the AES or BIOS without affecting the debugger. MonST2C will also work in low resolution, thus allowing you to debug programs that run in low resolution.

Initially we shall describe how to use MonST2C to debug programs that are compiled as a single module. Using it with multi-file applications will be described later.

If you are already an expert at using the version of MonST supplied with DevpacST version 2 then please read the next two pages on the use of MonST2C with Lattice C.

Preparing to use MonST2C

If you are going to debug a program using MonST2C you should ensure that you have selected the compiler's -d3 flag when compiling your program, either using the Compiler Options box or from the compiler's command line.

When linking you should either check Linker Symbols from the Options menu if invoking the linker from the editor or use the XADDSYM keyword if using an explicit linker command line or link file. Do not use the -Ln option as this will remove much of the debugging information.

This will ensure that the addresses of your functions, the library functions that you use, your program's external variables, and the address corresponding to each line of your source code will be stored in your executable file. Don't be surprised if this causes it to increase in size dramatically!

Whilst we recommend using -d3 for most purposes there are two other variants of the -d flag that you may find useful:

Using -d1 includes the name and line number information for MonST2C, producing very much smaller files than using -d3, but has two disadvantages compared with -d3.

Normally the compiler will make some of your variables register variables automatically. Using -d3 will force the compiler to store every such auto-generated register variable in memory at the end of each statement. This has the advantage of generally making the assembly language code generated easier to understand. If you are using the -d1 flag then it is probably a good idea to use the -mr compiler flag which will disable the automatic registerisation completely.

The other disadvantage of -d1 is that it only stores the filename of the source file in the debug information rather than the full pathname. Thus it can only be used to debug files that are in the current directory.

The -d2 option is exactly the same as -d3 except that the output files are slightly smaller and the compiler chosen register variables will not be flushed to memory; they will *remain* in registers.

Using -d4 and -d5 will create still larger files than -d2 and -d3, however the additional information is of no use to MonST2C.

Don't use the Global Optimiser when preparing code to be debugged with MonST2C as the transformations that it performs can make the code produced appear to bear very little similarity to your source code!

Invoking MonST2C

From the Desktop

MonST2C is supplied as a GEM program with extension .PRG; to debug a TOS application you can rename it as MONST2C.TOS or install it as a TOS program. This will ensure that the operating system will perform the same initialisation as if you were running your program without it. Once executed MonST2C will prompt you for the name of the file to load.

If you debug a TOS program with the GEM version of the debugger it will work fine but the screen display will probably be messy; however, debugging a GEM program with a TOS debugger will cause all sorts of nasty problems to occur and should be avoided.

From the Editor

If you are using the standard editor configuration file, EDCTOOLS.INF, MonST2C can be invoked using the Debug option on the Tools menu and Alt and the 5 key on the numeric key pad. Of course, the editor will need to be able to find the MONST2C.PRG file for this to work. Invoking the debugger in this way will load the .PRG file corresponding to the file being edited.The debugger will also load your source file too.

The type of initial screen mode used when invoked from the editor is determined by the GEM and TOS buttons in the Tool Configuration dialog. See the section **EdC, The Screen Editor** for more details. The rules described above about using the wrong type of screen initialisation are also relevant here.

From Batcher

If you wish to invoke MonST2C from Batcher, just type

```
monst2c test
```

if test is the program that you are debugging.

MonST2C Dialog and Alert Boxes

MonST2C makes extensive use of dialog- and alert-boxes which are similar in concept to those used by GEM programs but have several differences. MonST2C does not use genuine GEM-type boxes in order for it to remain *robust* - that is to avoid interaction when debugging programs that themselves use GEM calls. In addition the mouse is not available within the debugger itself which makes objects like true GEM buttons impossible.

A MonST2C dialog box displays the prompt ESC to abort above the top left corner of the box together with a prompt, normally followed by a blank line with a cursor. At any time a dialog box may be aborted by pressing Esc, or data may be entered by typing. The cursor, Backspace and Del keys may be used to edit entered text in the usual way and the whole line may be deleted by pressing the Clr key - note that this is different to GEM dialog boxes which use the Esc key to delete a whole line of text. An entered line is terminated by pressing the Return key, though if the line contains errors the screen will flash and the Return key will be ignored allowing correction of the data before pressing Return again. Another difference is that dialog boxes that require more than one line of data to be entered do not allow the use of the cursor up and down keys to switch between different lines - in MonST2C the lines have to be entered in order.

A MonST2C alert box is a small box displaying a message together with the prompt [Return] and is normally used to inform the user of some form of error. The box will disappear on pressing the Return or Esc keys, whichever is more convenient.

Initial Display

If you have run MonST2C without a command line you will be presented with a dialog box prompting for an executable program name. You should enter the name of the program that you wish to debug. If you omit the file's extension, MonST2C will look for a .PRG, .TTP and .TOS file in that order.

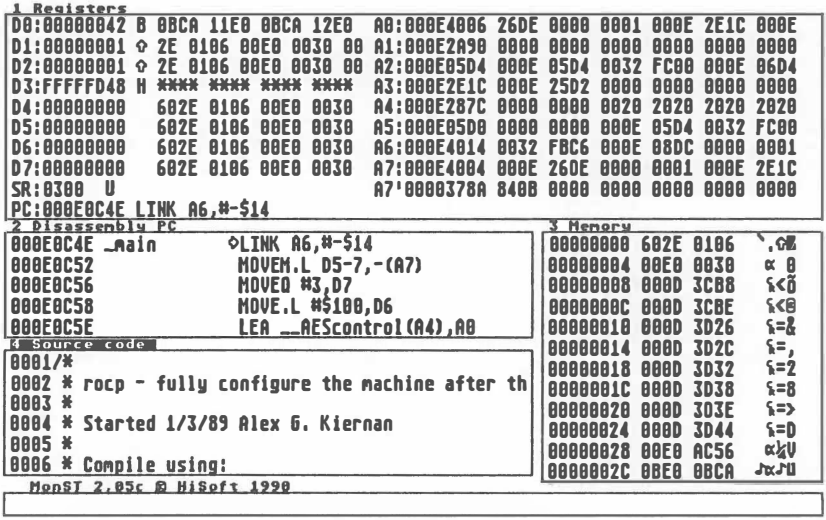
Certain features work differently or are not available when using MonST2C in low resolution. They are shown with this icon.

Front Panel Display

The main display of MonST2C is via a *Front Panel* showing registers, memory and instructions. The name Front Panel stems from the type of panels that were mounted on mainframe and mini computers to provide information on the state of the machine at a particular moment, usually through the use of flashing lights. These lights represent whether or not particular flip-flops (electronic switches) within the computer are open or closed; the flip-flops that are chosen to be shown on this panel are normally those that make up the internal registers and flags of the computer thus enabling programmers and engineers to observe what the computer is doing when running a program.

So these are *hardware* front panel displays; what MonST2C provides you with is a *software* front panel - the code within MonST2C works out the state of your computer and then displays this information on the screen.

The initial MonST2C display consists of five windows, similar to those shown below. In low-resolution the arrangement of the windows is slightly different to allow efficient use of the smaller available screen space.



The top window (1 Registers A4) displays the values of the machine's data and address registers, together with the memory pointed to by these registers.

The next window (2 Disassembly PC) is the disassembly window; this displays several lines of assembly instructions, by default based around the program counter (PC), shown in the title area of the window. A ⇒ sign is used to denote the current value of the PC.

Window number 3 is the memory window which displays a section of memory in word-aligned hex and ASCII.

Window number 4 is the source code window; it shows a a portion of your source code and the corresponding line numbers.

The final window at the bottom of the screen, which is un-numbered, is the smallest window and is used to display messages.

One of the most powerful features of MonST2C is its flexibility with windows - an extra window may be created, the font size can be changed, and windows may be locked to particular registers; these features are detailed later.

Simple Window Handling

MonST2C has the concept of a *current window* - this is denoted by displaying its title in black. The current window may be changed by pressing the Tab key to cycle between them, or by pressing the Alt key together with the window number, for example Alt-2 selects the disassembly window. (AZERTY keyboard users please note - the Shift key is *not* required when using Alt to select windows). Note that the lowest window can never be made the current window - it is used solely for displaying messages.

Command Input

MonST2C is controlled by single-key commands which creates a very fast user-interface, though this can take getting used to if you are familiar with a line-oriented command interface of another debugger. Users of HiSoft DevpacST and HiSoft debuggers on other machines should find that they are already familiar with many of the commands.

In general the Alt key is the window key - when used in conjunction with other keys it acts on the *current window.*

Commands may be entered in either upper or lower case. Those commands whose effects are potentially disastrous require the Ctrl key to be pressed in addition to a command key. The keys used were chosen to be easy to remember, wherever possible. Commands take effect immediately - there is no need to press Return - and invalid commands are simply ignored. The relevant sections of the front panel display are updated after each command so any effects can be seen immediately.

MonST2C is a powerful and sometimes complex program and we realise that it is unlikely that many users will use every single command. For this reason the remainder of the MonST2C manual is divided into two sections - the former is an introduction to the basic commands of the program, while the latter is a full reference section. It is possible for new users and beginners to use the debugger effectively while having only read the **Overview**; don't be intimidated by the **Reference** section.

MonST2C Overview

The most common low-level command in MonST2C is probably single-step, obtained by pressing Ctrl-Z (or Ctrl-Y if you find it more convenient). This will execute the instruction at the PC, the one shown in the Register window and, normally, also in the Disassembly window. After executing it the debugger re-displays the values of the registers and memory displayed, so you can watch the processor execute your program, step by step. Single-stepping is the best way of going through sections of code where you don't understand what is going on, but it is also the slowest - and it deals with your program only on an assembly language level, not in terms of your C program. There is, of course, an answer.

A *breakpoint* is a special word placed into your program to stop it running and enter MonST2C. There are many types of breakpoint but we will restrict ourselves to the simplest for now. A breakpoint may be set by pressing Alt-B, then entering the address you wish to place the breakpoint. You can enter an address in MonST2C as a symbol, as a hexadecimal line number preceded by #, hex (the default base), a decimal line number preceded by #\ or as a complex expression. Examples of valid addresses are main, foo, #10, #\123 10+mydata. If you type in an invalid address the screen will flash and allow you to correct the expression. You can omit the leading _ or @ of C function names if you wish.

Having set a breakpoint you need some way of letting your program actually run, and Ctrl-R will do this. If will execute your program using the registers displayed and starting from the PC. MonST2C will be re-entered if a breakpoint has been hit, or if a processor exception occurs.

MonST2C uses its own screen display which is independent from your programs. If you press the V key you will see your current programs display, pressing another key switches you back to MonST2C. This allows you to debug programs without disturbing their output at all.

Any window may be zoomed to the full screen size by pressing Alt-Z. To return to the main display press Alt-Z or the Esc key. The Esc key is also the best way of getting out of anything you may have invoked by accident. The Zoom command, like all Alt-commands, works on the *current window* which you can change by pressing Tab. You can dump the current window to your printer by pressing Alt-P.

To change the address from which a window displays its data, press Alt-A, then enter the new address. Note that the disassembly window will always re-display from the PC after you single-step, because it is *locked* to the PC. The locking of windows is detailed in the Reference section.

To quit MonST2C press Ctrl-C. Strange as it may sound this will not always work - what Ctrl-C does is terminate the *current program*, which may be MonST2C or, more likely, the program you are debugging., in which case MonST2C will still be in control. You know when you have terminated the program under investigation because it will say so in the lower window. Once your program has been terminated, pressing Ctrl-C will terminate MonST2C.

We hope this overview has given you a good idea of the most common features of MonST2C to let you get on with the complex process of writing and debugging programs. When you feel more confident you should try and read the **Reference** section, probably best taken, like all medicine, in small doses.

MonST2C Reference

Numeric Expressions

MonST2C has a full expression evaluator, based on that in the DevpacST assembler, GenST, including operator precedence. We decided that changing MonST2C to use the standard C operators would be confusing for users who are already familiar with MonST2.

The following operators are supported, in decreasing order of precedence:

monadic minus (-) and plus (+), address of line number (#)

bitwise not (~)

shift left (<<) and shift right (>>)

bitwise And (&), Or (!) and Xor (^)

multiply (*) and divide (/)

addition (+) and subtraction (-)

equality (=), less than (<), greater than (>), not equals (<>)

The comparison operators are signed and return 0 if false or -1 ($FFFFFFFF) if true. The shift operators take the left hand operand and shift it the number of bits specified in the right hand operand, vacated bits are filled with zeroes.

This precedence can be overridden by the use of parentheses (and). With operators of equal precedence, expressions are evaluated from left-to-right. Spaces in expressions (other than those within quotes - ASCII constants) are not allowed.

All expression evaluation is done using 32-bit signed-integer arithmetic, with no checking of overflow.

The MonST2C expression evaluator also supports indirection using the { and } symbols. Indirection may be performed on a byte, word or long basis, by following the } with a period then the required size, which defaults to long. If the pointer is invalid, either because the memory is unreadable or not an even address (if word or longword indirection is used) then the expression will not be valid.

For example, the expression

```
{data_start+10}.w
```

will return the word contents of location data_start+10, assuming data_start is even. Indirection may be nested in a similar way to ordinary parentheses.

Numbers

Absolute numbers may be in various forms:

decimal constants, e.g. \1029

hexadecimal constants, e.g. 12f or $12f

octal constants, e.g. @730

binary constants, e.g. %1100010

character constants, e.g. 'X'

\ is used to denote decimal numbers, $ is used to denote hexadecimal numbers (the default), % for binary numbers, @ for octal numbers and single ' or double " quotes for character constants.

Character Constants

Whichever quote is used to mark the start of a string must also be used to denote its end and quotes themselves may be used in strings delimited with the same quote character by having it occur twice. Character constants can be up to 4 characters in length and evaluate to right-justified longs with null-padding if required. For example, here are some character constants and their ASCII and hex values:

"Q"	Q	$00000051
'hi'	hi	$00006869
"Test"	test	$54657374
"it's"	it's	$6974277C
'it''s'	it's	$6974277C

Symbols and Registers

Symbols may be referred to and are normally case-sensitive and significant to either 8 or 22 characters (depending on the form of debug information used), though this can be changed with Preferences.

Normally there is no need to type the initial _ or @ of C language labels. You can enforce the need to include the _ or @ with the Preferences command

Registers may be referred to simply by name, such as A3 or D7 (case insensitive), but this clashes with hex numbers. To obtain such hex numbers precede them with either a leading zero or a $ sign. A7 refers to the *user* stack pointer.

There are several reserved symbols which are case insensitive, namely TEXT, DATA, BSS, END, SP, SR, and SSP. END refers to one byte past the end of the BSS section and SP refers to either the user- or supervisor-stack, depending on the current value of the status register. Remember that the names of all your external variables and functions will also be available.

In addition there are 10 memories numbered M0 through M9, which are treated in a similar way to registers and can be assigned to using the Register Set command. Memories 2 through 5 inclusive refer to the current start address of the relevant window and modifying them will change the start address of that window.

Window Types

There are four window types and the exact contents of these windows and how they are displayed is detailed below. The allowed types of windows are shown in the table below.

Window	Allowed Types
1	Register
2	Disassembly
3	Memory
4	Disassembly, Memory or Source code
5	Memory

Register Window Display

The data registers are shown in hex, together with the ASCII display of their low byte and then a hex display of the eight bytes they point to in memory. The address registers are also shown in hex, together with a hex display of 12 bytes. As with all hex displays in MonST2C this is word-aligned, with non-readable memory displayed as **.

The status register is shown in hex and in flag form, additionally with U or S denoting user- or supervisor-modes. A7' denotes the supervisor stack pointer, displayed in a similar way to the other address registers.

The PC value is shown together with a disassembly of the current instruction. Where this involves one or more effective addresses these are shown in hex, together with a suitably-sized display of the memory they point to.

For example, the display

```
TST.W $12A(A3) ;00001FAE 0F01
```

signifies that the value of $12A plus register A3 is $1FAE, and that the word memory pointed to by this is $0F01. A more complex example is the display

```
MOVE.W $12A(A3),-(SP) ;00001FAE 0F01 ⇒0002AC08 FFFF
```

The source addressing mode is as before but the destination address is $2AC08, presently containing $FFFF. Note that this display is always of a suitable size (MOVEM data being displayed as a quad-word) and when pre-decrement addressing is used this is included in the address calculations.

No hex data is shown for the data registers and the address register data area is reduced to 4 bytes. In addition the disassembly line may not be long enough to display complex addressing modes such as the second example above.

Disassembly Window Display

Disassembly windows display memory as disassembled instructions. On the left the hex address is shown, followed by any symbol, then the disassembly itself. The current value of the PC is denoted with ⇒.

If the instruction has a breakpoint placed on it this is shown using square brackets ([ ]) afterwards, the contents of which depend on the type of breakpoint. For stop breakpoints this will be the number of times left for this instruction to execute, for conditional breakpoints this will be a ? followed by the beginning of the conditional expression, for count breakpoints this will be a = sign followed by the current count, and for permanent breakpoints a * is shown.

The exact format of the disassembled op-codes is Motorola standard, as accepted by the assembler, asm. All output is upper-case (except lower-case labels) and all numeric output is hex, except trap numbers. Leading zeroes are suppressed and the $ hex delimiter is not shown on numbers less than 10. Where relevant numerics are shown signed. The only deviation from Motorola standard is the register lists shown in MOVEM instructions - in order to save display space the type of the second register in a range is abbreviated, for example

```
MOVEM.L d0-d3/a0-a2,-(sp)
```

will be disassembled as

```
MOVEM.L d0-3/a0-2,-(sp)
```

Any displayed symbols replace the hex address display, limited to a maximum of 8 characters.

Memory Window Display

Memory windows display memory in the form of a hex address, word-aligned hex display and ASCII. Unreadable memory locations are denoted by **. The number of bytes shown is calculated from the window width, up to a maximum of 16 bytes per line.

Source-code Window Display

The source-code window displays ASCII files in a similar way to a screen editor. The default tab setting is 8 though this can be toggled to 4 with the Edit Window command.

Window Commands

The Alt key is generally used for controlling windows, and when used applies to the *current window*. This is denoted by having an inverse title and can be changed by pressing Tab, or Alt and the window number.

Most window commands work in any window, zoomed or not, though when it does not make sense to do something the command is ignored.

Alt-A — Set Address

This sets the starting address of a memory or disassembly window.

Alt-B — Set Breakpoint

Allows the setting of any type of breakpoint, described later under **Breakpoints**.

Alt-E — Edit Window

On a memory window this lets you edit memory in hex or ASCII. Hex editing can be accomplished using keys 1-9, A-F, together with the cursor keys. Pressing Tab switches between hex & ASCII, ASCII editing takes each keypress and writes it to memory. The cursor keys can be used to move about memory. To leave edit mode press the Esc key.

On a register window this is the same as Alt-R, Register Set, described shortly.

On a source-code window this toggles the tab setting between 4 and 8.

Alt-F — Font size

This changes the font size in a window. In high resolution 16 and 8 pixel high fonts are used, in colour 8 and 6 pixel high fonts are used. This allows a greater number of lines to be displayed, assuming your monitor can cope.

Changing the font size on the register window causes the position of windows 2 and 3 to be re-calculated to fill the available space.

Alt-L — Lock Window

This allows disassembly and memory windows to be locked to a particular register. After any exception the start address of the window is re-calculated, according to the locked register.

To unlock a window simply enter a blank string.

By default window 2 is locked to the PC. You can lock windows to each other by specifying a lock to a memory window, such as M2.

Alt-O — Show Other

This prompts for an expression and displays it in hex, decimal and as a symbol if relevant.

Alt-P — Printer Dump

Dumps the current window contents onto the printer. The print can be aborted by pressing Esc.

Alt-R — Register Set

Allows any register to be set to a value, by specifying the register, an equals sign, then its new value. It can also be used to set the values of the MonST2C memories M0 to M9. For example the line

```
a3=a2+4
```

sets register A3 to be A2 plus 4. You can also use this to set the start address of windows when in zoom mode so that on exit from zoom mode the relevant window starts at the required address.

Do not assign to M4 if window 4 is currently a source-code window.

Alt-S — Split windows

This either splits window 2 into window 2 and window 4, or splits window 3 into window 3 and window 5. Each new window is independent from its creator. Pressing Alt-S again will unsplit the window.

This command has no effect.

Alt-T — Change Type

This only works on window 4 (created either by splitting window 2 or by loading a source file). It changes the type of the window between disassembly, memory and source-code (if a file has been loaded).

Alt-Z — Zoom Window

This zooms the current window to be full size. Other Alt commands are still available and normal size can be achieved by pressing Esc or Alt-Z again.

Zooming the register window can be extremely useful as it allows you to see the 'hidden' registers M0, M1 and M6-M9.

Cursor Keys

The cursor keys can be used on the current window, and their action depends on the window type.

On a memory window all four cursor keys change the current address, and Shift ↑ and Shift ↓ move a page in either direction.

On a disassembly window ↑ and ↓ change the start address on an instruction basis, ← and → change the address on a word basis.

On a source-code window ↑ and ↓ change the display on a line basis, and Shift ↑ and Shift ↓ on a page basis.

Screen Switching

MonST2C uses its own screen display and drivers to prevent interference with a program's own screen output. To prevent flicker caused by excessive screen switching when single-stepping the screen display is only switched to the program's after 20 milliseconds, producing a flicker-free display while in the debugger. In addition the debugger display can have a different screen resolution to your program's if using a colour monitor.

V — View Other Screen

This flips the screen to that of the programs, any key returns to the MonST2C display.

This changes the screen mode of MonST's display between low and medium resolution. It re-initialises window font sizes and window positions to that of the initial display. This will not effect the screen mode of the program being debugged.

This command is ignored on a monochrome monitor.

As MonST2C has its own idea of where the screen is, what mode it is in and what palette to use you can use MonST2C to actually look at the screen memory in use by your program, ideal for low-level graphics programs.

If your program changes screen position or resolution, via the XBIOS or the hardware registers, it is important that you temporarily disable screen switching using Preferences while executing such code; otherwise MonST2C will not notice the new attributes of your program's screen.

When a disk is accessed, when loading or saving, the screen display will probably switch to the program's during the operation. This is in case a disk error occurs, such as a write-protection violation or a read error, as it allows any GEM alert boxes to be seen and acted upon.

Breaking into Programs

Shift-Alt-Help Interrupt Program

While a program is running it can be interrupted by pressing this key combination, which will cause a trace exception at the current value of the PC. With computationally intensive program sections this will be within the program itself but with a program making extensive use of the ROM, such as the GEMDOS or AES, the interruption will normally be in the ROM itself, or the line-F handler stored in low-memory. If this is the case it is recommended that a breakpoint be placed in your actual program area then a Return to Program command (Ctrl-R) issued.

Pressing Alt-Help without the Shift key will normally produce a screen dump to the printer - if you press this accidentally it should be pressed again to cancel the dump.

It is possible for this key combination to be ignored when pressed - if this occurs press it again and it should work. Pressing it when in MonST2C itself will produce no effect.

A program should never be terminated (using Ctrl-C) if it has just been interrupted in the middle of a ROM routine. This is likely to cause a system crash.

Breakpoints

Breakpoints allow you to stop the execution of your program at specified points within it. MonST2C allows up to eight simultaneous breakpoints, each of which may be one of five types. When a breakpoint is hit MonST2C is entered and then decides whether or not to halt execution of your program, entering the front panel display, or continue; this decision is based on the type of the breakpoint and the state of your program's variables.

Simple Breakpoints

These are one-off breakpoints which, when executed, are cleared and cause MonST2C to be entered.

Stop Breakpoints

These are breakpoints that cause program execution to stop after a particular instruction has been executed a certain number of times. In fact a simple breakpoint is really a stop breakpoint with a count of one.

Count Breakpoints

Merely counters; each time such a breakpoint is reached a counter associated with it is incremented, and the program will resume.

Permanent Breakpoints

These are similar to simple breakpoints except that they are never cleared - every time execution reaches a permanent breakpoint MonST2C will be entered.

Conditional Breakpoints

The most powerful type of breakpoint which allow program execution to stop at a particular address only if an arbitrarily complex set of conditions apply. Each conditional breakpoint has associated with it an expression (conforming to the rules already described). Every time the breakpoint is reached this expression is evaluated, and if it is non-zero (i.e. true) then the program will be stopped, otherwise it will resume.

Alt-B — Set Breakpoint

This is a window command allowing the setting or clearing of breakpoints at any time. The line entered should be one of the following forms, depending on the type of breakpoint required:

<address>

will set a simple breakpoint.

<address>,<expression>

will set a stop breakpoint at the given address, after it has executed <expression> times.

<address>,=

will set a count breakpoint. The initial value of the count will be zero.

<address>,*

will set a permanent breakpoint.

<address>,?<expression>

will set a conditional breakpoint, using the given expression.

<address>,-

will clear any breakpoint at the given address.

Breakpoints cannot be set on addresses which are odd or unreadable, or in ROM, though ROM breakpoints may be emulated using the Run Until command.

Every time a breakpoint is reached, regardless of whether the program is interrupted or resumed, the program state is remembered in the History buffer, described later.

Help — Show Help and Breakpoints

This displays the text, data and BSS segment addresses and lengths, together with every current breakpoint. Alt-commands are available within this display.

Ctrl-B — Set Breakpoint

This sets a simple breakpoint at the start address of the current window, so long as it is a disassembly window. If a breakpoint is already there then it will be cleared.

U — Go Until

This prompts for an address, at which a simple breakpoint will be placed then program execution resumed.

Ctrl-K — Kill Breakpoints

This clears all set breakpoints.

Ctrl-A — Set Breakpoint then Execute

A command that places a simple breakpoint at the instruction *after* that at the PC and resumes execution from the PC. This is particularly useful for DBF-type loops if you don't want to go through the loop, but just want to see the result after the loop is over.

Ctrl-D — GEMDOS Breakpoint

This allows a breakpoint to be set on specific GEMDOS calls. The required GEMDOS number should be entered, or a blank line entered if any existing GEMDOS breakpoint needs to be cleared.

History

MonST2C has a *history buffer* in which the machine status is remembered for later investigation.

The most common way of entering data into the history buffer is by using you single-step but, in addition, every breakpoint reached and every exception caused enters the machine state into the buffer. Various forms of the Run command also cause entries to be made into this buffer.

The history buffer has room for five entries - when it fills up, the oldest entry is removed to make room for the newest entry.

H — Show History Buffer

This opens a large window displaying the contents of the history buffer. All register values are shown including the PC as well as a disassembly of the next instruction to be executed.

If a disassembly in the History display includes an instruction which has a breakpoint placed on it the [ ]s will show the *current* values for that breakpoint, not the values at the time of the entry into the history buffer.

Quitting MonST2C

Ctrl-C — Terminate

This will issue a terminate trap to the *current* GEMDOS task. If a program has been loaded from within MonST2C it will be terminated and the message Program Terminated appear in the lower window. Another program can then be loaded, if required.

If no program has been loaded into MonST2C it will itself terminate when this command is used.

Terminating some GEM programs prematurely, before they have closed workstations or restored window control properly can seriously confuse the AES and VDI. This may not be noticeable immediately but often causes crashes when a subsequent program is executed.

Loading & Saving

Ctrl-L — Load Executable Program

This will prompt for an executable filename then a command line and will attempt to load the file ready for execution. If MonST2C has already loaded a program it is not possible to load another until the former has terminated.

The file to be loaded must be an executable file - attempting to load a non-executable file will normally result in TOS error 66 and further attempts to load executable files will normally fail as GEMDOS does not de-allocate the memory it allocated before trying to load the errant file. If this occurs terminate MonST2C, re-execute it and use the Load Binary File command.

B — Load Binary File

This will prompt for a filename and optional load address (separated by a comma) and will then load the file where specified. If no load address is given, memory will be allocated from GEMDOS and used. M0 will be set to the start address and M1 to the end address.

S — Save Binary File

This will prompt for a filename, a start address and an (inclusive) end address. To re-save a file recently loaded with the Load Binary File command <filename>,M0, and M1 may be specified, assuming of course that M0 and M1 have not been re-assigned.

A — Load ASCII File

This powerful command allows an ASCII file, normally of source code, to be loaded and viewed within MonST2C. Window 4 will be created if required then set up as a source-code window. Memory for the source code is taken from GEMDOS so sufficient free memory must be available. It is recommended that source-code be loaded *before* an executable program to ensure enough memory.

If the file currently being debugged contains Lattice debug information, MonST2C will use the line number information corresponding to the new source file that has been loaded. Thus, loading the source file will change the effect of the # operator.

Window 4 is not shown, though an ASCII file may may be loaded in low-res then viewed after switching to medium resolution using Ctrl-O and pressing Alt-S, Alt-T, Alt-T.

If an ASCII file is loaded *after* an executable program the memory used will be owned by the *program itself*, not MonST2C. When such a program terminates, any displayed source-code window will be closed. This is also the case when the source text is automatically loaded by MonST2C.

Executing Programs

Ctrl-R — Return to program / Run

This runs the current program with the given register values at full speed and is the normal way to resume execution after entry via a breakpoint.

Ctrl-Z — Single-Step

This single-steps the instruction at the PC with the current register values. Single-stepping a TRAP, Line-A or Line-F opcode will, by default, be treated as a single instruction. This can be changed using Preferences.

Ctrl-Y — Single-Step

Identical to Ctrl-Z above but included for the convenience of German users.

Ctrl-T — Interpret an Instruction (Trace)

This interprets the instruction at the PC using the displayed register values. It is similar to Ctrl-Z but skips *over* BSRs, JSRs, TRAPs, Line-A and Line-F calls, re-entering the debugger on return from them to save stepping all the way through the routine or trap. It works on instructions in ROM or RAM.

Ctrl-S — Skip an Instruction

Ctrl-S increments the PC register by the size of the current instruction thus causing it to be skipped. Use this instead of Ctrl-Z when you know that this instruction is going to do something it shouldn't or that you don't like.

R — Run (various)

This is a general Run command and prompts for the type of execution, selected by pressing a particular key.

Run G Go

This is identical to Ctrl-R, Run, and resumes the program at full speed.

Run S Slowly

This will run the program at reduced speed, remembering every step in the history buffer.

Run I Instruction

This is similar to Run Slowly but allows a count to be entered, so that a particular number of instructions may be executed before MonST2C is entered.

Run U Until

You will be prompted for an expression which will be evaluated after every instruction. The program will then run, albeit at reduced speed, until the given expression evaluates to non-zero (true) when MonST2C will be entered. For example if single-stepping a DBF loop which used d6 in the ROM code you could say Run Until d6&ffff=ffff (waiting for the low word of d6 to be $FFFF) or, alternatively, PC=FC8B1A, or whatever.

This should not be confused with the Until command, which takes an address, places a breakpoint there then resumes execution.

With all of these commands (except Run Go) you will then be asked Watch Y/N? If Y is selected then the MonST2C display will be shown after every instruction and you can watch registers and memory as they change, or interrupt execution by pressing both Shift keys simultaneously. If N is selected then execution will occur while showing your program's display and execution may be interrupted by pressing Shift-Alt-Help.

Selecting Watch mode with screen switching turned off is likely to result in a great deal of eye strain as the display will be flipped after each and every instruction, particularly alarming with colour monitors.

With any of these Run modes (except Go) all information after every instruction will be remembered in the history buffer. In addition TRAPs will be treated as single-instructions, unless changed with Preferences; though see the warnings under that command about tracing all the way through ROM routines.

When a program is running with one of the above modes a couple of pixels near the top left of the display will flicker, to denote that something is happening, as it is possible to think the machine has hung when, in fact, it is simply taking a while to Run through the code, an instruction at a time.

Searching Memory

G search memory (Get a sequence)

This will prompt Search for B/W/L/T/I?, standing for Bytes, Words, Longs, Text and Instructions.

If you select B, W or L you will then be prompted to enter the sequence of numbers you wish to search for, each separated by commas. MonST2C is not normally fussy about word-alignment when searching, so it can find longs on odd boundaries, for example. If you wish to force a particular alignment, finish the list of items to search for with ',W'; for word boundaries or ',L' for longword boundaries.

If you select T you may search for any given text string, for which you will be prompted. You will also be asked whether you wish the search to be case sensitive; if you press Y then Test will match TEST or TeSt

If you select I you can search for part or all of the mnemonic of an instruction; for example if you searched for $14(A you would find an instruction like MOVE.L D2,$14(A0). The case of the string you enter *is* important (unlike MonST version 1), but you should bear in mind the format the disassembler produces, e.g. always use hex numbers, refer to A7 rather than SP and so on.

Once you have selected the search type and parameters, the search begins, control passing to the Next command, described below.

N — find next

This can be used after the G command to find subsequent occurrences of the search data. With the B, W, L and T options you will always find at least one occurrence, which will be in the buffer within MonST2C that is used for storing the sequence. With the T option you may also find a copy in the system keyboard buffer. With these options, the Esc key is tested every 64k bytes and can be used to stop the search. With the I option, which is very much slower, the Esc key is tested every 2 bytes.

The search area of memory goes from 0 to the end of RAM, then via the system ROM area and cartridge area then back to 0. MonST2C will not search the cartridge area if the environment variable NOCARTRIDGE exists.

The search will start just past the start address of the current window (except register windows) and, if an occurrence is found, it will re-display the window at the given address.

Searching Source-Code Windows

If the G command is used on a source-code window the T sub-command is chosen automatically and, if the text is found, the window will re-display the line containing it.

Miscellaneous

Ctrl-P — Preferences

This permits control over various options within MonST2C. The first three require Y/N answers, pressing Esc aborts and Return leaves them alone.

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.

If you are interested in this sort of low-level hackery, you should consider purchasing DevpacST as it provides facilities from recovering from the after effects of interrupting the operating system code.

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.

Ignore 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.

Show Line Numbers in Source

MonST2C can either show line numbers in your source window in decimal (press D), hexadecimal (H) or not at all (press N). Using hexadecimal line numbers has the advantage that you can use them directly with the # line number operator. This if you can see that you want to execute your program until the line with number 001C then just type U (for run until) #1C. Remember how ever that A0 through A7 and D0 through D7 are register names and take priority over hexadecimal numbers. To enter line number A0 use #$A0.

Decimal line numbers are naturally more civilised but remember that you need to prefix any decimal number with \. If you want to find the address of line 28 decimal, use #\28 not #28.

Auto Load Source

Using the default settings, MonST2C 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. MonST2C loads the source file corresponding to the first module with debug information in the file that you are debugging. This would normally be your main program. You can disable this feature if you do not wish to load this source file or you wish to debug a program written in another language. Please see the sections at the end of this chapter concerning the use of MonST2C with multi-module programs.

Automatic Prefix Labels

Using the default setting MonST2C 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, the MonST2C will try _main or if this doesn't exist then it will try @main.

This facility is extremely useful since C functions normally have an _ added by the compiler. When using register passing @ is used as the prefix instead of _. Thus you can just use the C name without bothering about the prefix.

Should there be an assembly language name the same as a C name, say test and _test, then you will need to use the _ explicitly to get the C function rather than the assembly language one.

You can disable this option so that only exact matches of names are supported.

Symbols Option

This allows control over the use of symbols in expressions in MonST2C. It will firstly ask whether the case of symbols should be ignored, pressing Y will cause case independent searching to be used. It will then prompt for the maximum length of symbols, which is normally 22 but may be reduced to as low as 8.

Top Of RAM

This indicates to MonST2C which memory location should be considered the top of memory by the Search Memory (G) command. Normally you will not need to change this as it defaults to the system variable phys_top; but you may need to modify it if you are debugging software that lowers phys_top.

Save preferences

Reply Y to this command to save your current preferences to the file MONST2.INF in the current directory. When MonST2 loads it will read your current preferences from this file. MONST2.INF must be in the current directory when MonST2C is loaded.

I — Intelligent Copy

This copies a block of memory to another area. The addresses should be entered in the form

```
<start>,<inclusive_end>,<destination>
```

The copy is intelligent in that the block of memory may be copied to a location which overlaps its previous location.

No checks at all are made on the validity of the move; copying to non-existent areas of memory is likely to crash MonST2C and corrupting system areas may well crash the machine.

L — List Labels

This opens up a large window and displays all loaded symbols. Any key displays the next page, pressing Esc aborts. The symbols will be displayed in the order they were found on disk.

W — Fill Memory With

This fills a section of memory with a particular byte. The range should be entered in the form

```
<start>,<inclusive_end>,<fillbyte>
```

The warning described under the I command about the lack of checks applies equally to this command.

M — Modify Address

Included for compatibility with MonST1, equivalent to Alt-A.

O — Show Other Bases

Included for compatibility with MonST1, equivalent to Alt-O.

D — Change Drive & Directory

This allows the current drive and sub-directory to be changed.

Ctrl-Alt-Numeric Dot — Reset machine

Holding down Ctrl and Alt and then pressing the Dot (.) key on the numeric keypad will cause the machine to be reset. Great for Mega ST owners with 1.2 ROMs but without long arms!

Ctrl-E — Re-Install Exceptions

This command causes MonST2C to re-install the exception vectors; useful if you are debugging a high level language program whose runtime routines use the exceptions. Naturally, Lattice C 5 programs do not normally modify the exception vectors. This must be used *after* the user's program has modified the exceptions.

Command Summary

Window Commands

Alt-A ..... Set Address
Alt-B ..... Set Breakpoint
Alt-E ..... Edit Window
Alt-F ..... Font Size
Alt-L ..... Lock Window
Alt-O ..... Show Other
Alt-P ..... Printer Dump
Alt-R ..... Register Set
Alt-S ..... Split Windows
Alt-T ..... Change Type
Alt-Z ..... Zoom Window

Screen Switching

V ..... View Other Screen
Ctrl-O ..... Other Screen Mode

Breakpoints

Alt-B ..... Set Breakpoint
Help ..... Show Help and Breakpoints
Ctrl-B ..... Set Breakpoint

U ..... Go Until
Ctrl-K ..... Kill Breakpoints
Ctrl-A ..... Set Breakpoint then Execute
Ctrl-D ..... GEMDOS Breakpoint

Loading and Saving

Ctrl-L ..... Load Executable Program
B ..... Load Binary File
S ..... Save Binary File
A ..... Load ASCII File

Executing Programs

Ctrl-R ..... Return to program / Run
Ctrl-Z ..... Single-Step
Ctrl-Y ..... Single-Step
Ctrl-T ..... Interpret an Instruction (Trace)
Ctrl-S ..... Skip Instruction
R ..... Run (various)

Searching Memory

G ..... Search Memory (Get a sequence)
N ..... Find Next

Miscellaneous

Ctrl-Alt-Dot ..... Reset machine
Ctrl-C ..... Terminate
Ctrl-E ..... Re-install breakpoints
Ctrl-P ..... Preferences
D ..... Change Drive & Directory
H ..... Show History Buffer
I ..... Intelligent Copy
L ..... List Labels
M ..... Modify Address
O ..... Show Other Bases
W ..... Fill Memory With
Shift-Alt-Help ..... Interrupt Program

Debugging Stratagem

Hints & Tips

If you have interrupted a program using Shift-Alt-Help or by a Run Until command and have found yourself in the middle of the ROM, there is a way of returning to the exact point in your program which called the ROM. Firstly ensure the Follow Traps option is on, then do Run Until with an expression of sp=a7. This will re-enter MonST2C the moment user mode is restored which will be in your program.

When using Run Until knowing that it will take a quite a while for the condition to be satisfied, give MonST2C a hand by pre-computing as much of the expression as you can, for example

```
(a3>(3A400-\100+M1))
```

could be reduced to

```
a3>xxx
```

where xxx has been calculated by you using the Alt-O command.

If you do use a label with Run Until then explicitly including any leading _ or @ will speed up the table search considerably.

Bug Hunting

There are probably as many strategies for finding bugs as there are programmers; there is really no substitute for learning the hard way, by experience. However, here are some hints which we have learnt, the hard way!

Firstly, a very good way of finding bugs is to look at the source code and think. The disadvantage of reaching first for the debugger, then second for the source code, is that it gets you into bad habits. You may switch to a machine or programming environment that does not offer debugging, or at least not one as powerful you are used to.

If a program fails in a very detectable way, such as causing an exception, debugging is normally easier than if, say, a program sometimes doesn't quite work exactly as it should.

Many bugs are caused by a particular memory location being stepped on. Where the offending memory location is detectable, by producing a bus error, for example, then a conditional breakpoint placed at one or more main subroutines can help greatly. For example, suppose the global variable main_ptr is somehow becoming odd during execution. The conditional expression could be set up as

```
{main_ptr}&1
```

If this method fails, and the global variable is being corrupted somewhere un-detectable, the remaining solution is to Run Until that expression, which could take a considerable time. Even then it may not find it, for example if the bug is caused by an interrupt happening at a certain time when the stack is in a particular place.

Count breakpoints are a good way of tracking down bugs *before* they occur. For example, suppose a particular subroutine is known to eventually fail but you cannot see why, then you should set a count breakpoint on it, then let the program run. At the point where the program stops, because of an exception say, look at the value of the count breakpoint (using Help). Terminate the program, re-load it, then set a stop breakpoint on the subroutine for that particular value or one before it. Let it run and then you can follow through the subroutine on the very call that it fails, to try and work out why.

Exceptions

MonST2C uses the 68000 processor exceptions to stop runaway programs and to single-step, so at this point it would be useful to explain them and what normally happens when they occur on an ST.

There are various types of exception that can occur, some deliberately, others accidentally. When one does occur the processor saves some information on the SSP, goes into supervisor mode and jumps to an exception handler. When MonST2C is active it re-directs some of these exceptions so it can take control when they occur. The various forms of exceptions, their usual results, and what happens when they occur with MonST2C active is shown in the following table:

Number	Exception	Usual effect	MonST2C active
2	bus error	bombs	trapped
3	address error	bombs	trapped
4	illegal instruction	bombs	trapped
5	zero divide	bombs	trapped
6	CHK instruction	bombs	trapped
7	TRAPV instruction	bombs	trapped
8	privilege violation	bombs	trapped
9	trace	bombs	used for single-stepping
10	line 1010 emulator	fast VDI interface	fast VDI interface
11	line 1111 emulator	internal TOS	internal TOS
32	TRAP #0	bombs	trapped
33	TRAP #1	GEMDOS call	GEMDOS call
34	TRAP #2	AES/VDI call	AES/VDI call
35-44	TRAP #3-#12	bombs	trapped
45	TRAP #13	XBIOS call	XBIOS call
46	TRAP #14	BIOS call	BIOS call
47	TRAP #15	bombs	trapped

Exceptions 2 to 8 are caused by a programmer error and are trapped by MonST2C.

Exception 9 can remotely be caused by programmer error and is used by MonST2C for single stepping.

Exceptions 10, 11, 33, 34, 45 and 46 are used by the system and left alone.

The rest (i.e. the unused TRAP exceptions) are diverted into MonST2C, but can subsequently be re-defined to be exploited by programs if required.

The 'bombs' entry in the table above means that the ST will attempt to recover from the exception, but it is not always successful.

When an exception occurs, the ST prints on the screen a number of *bomb* shapes (or *mushrooms* on the old disk-loaded TOS), the number being equal to the exception number. Having done this, it will abort the current program (losing any unsaved data from it) and attempt a return to the Desktop.

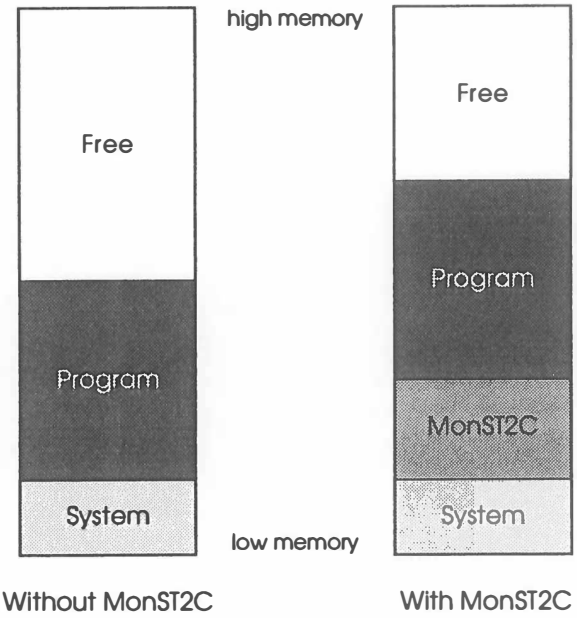
If the exception was caused by or resulted in important system variables being destroyed then the attempt may fail and the machine will not recover.

Occasionally very nasty crashes can cause the whole screen to fill with bombs (or mushrooms) which looks very impressive, but is not very useful!

Memory Layout

The usual versions of MonST2C co-reside with programs being debugged; that is, they are loaded, ask for a filename, and load that file in together with any labels.

It is useful to examine the usual logical memory map both with and without MonST2C, shown below:



The actual code size of MonST2C is around 25k, but in addition it requires an additional 32k of workspace. This may seem large but it is required for the copy of the ST screen memory saved by MonST2C; this is a most useful feature of the debugger.

Exception Analysis

When an unexpected exception occurs, it can be useful to be able to work out where and why it occurred and, possibly, to resume execution. Often a quick curse and a look at your source code may get your program working quicker though!

Bus Error

If the PC is in some non-existent area of memory then look at the relevant stack to try and find a return address to give a clue as to the cause. If the PC is in a correct area of your program then the bus error must have been caused by a memory access to non-existent or protected memory. Recovering from bus errors and resuming execution is generally not possible.

Address Error

If the PC is somewhere strange the method above should be used, otherwise the error must have been caused by a program access to an odd address. Correcting a register value may be enough to resume execution, at least temporarily.

Illegal Instruction

If the PC is in very low memory, below around $30, it is probable that it was caused by a jump to location 0. If you use MonST2C to look here you will see a short branch together with, normally, various ORI instructions (really longword pointers) and eventually an illegal instruction.

Privilege Violation

This is caused by executing a privileged instruction in user mode, normally meaning your program has gone horribly wrong. Bumping the PC past the offending instruction is unlikely to be much help in resuming the program.

Using MonST2C with other languages

A major feature of MonST2C is its ability to use symbols taken from the original program whilst debugging. MonST2C supports two formats for label information - the DRI standard, which allows up to 8 characters per symbol, and the HiSoft Extended Debug format, allowing up to 22 characters.

Most of HiSoft's language products support both formats (for example, DevpacST, HiSoft BASIC and FTL Modula-2) and many other vendors' compilers and linkers have an option to produce DRI-format debugging information.

The line number information format is, at the present time, specific to Lattice C and as such the line number operator, #, can only be used with programs compiled with one of the debugging options (-d1 to -d5).

Using MonST2C with multi-module programs

MonST2C will initially read the line number information and source file for the first module in the file that was compiled using -d. Thus, if you are only interested in debugging one module at a time then compile just this module with -d3: the appropriate source code will be loaded automatically.

If you wish to debug more than one file at once, then you can switch to another file by explicitly loading the appropriate source file using the A command.

For Devpac MonST2 Users

If you are used to the version of MonST that is part of Devpac ST version 2, here are the differences:

- Source line numbers can now be displayed in either hexadecimal or decimal. This is set using the Preferences command.
- Using the default settings, MonST2C will automatically run the program until the label _main and load the source file corresponding to the first debug information in the file.
- The operator # is used to give the address corresponding to a given C line number. To use this you need to use the compiler's -d option. The argument to # is a general MonST2C expression and so when using a number this should be in hexadecimal or prefixed by \ for decimal. Thus #10 and #\16 both give the address of line 16 of the program.
- The ASCII load command will change the action of the # operator if the appropriate debug information is available.
- MonST2C has no support for disassembly to disk. Dissassembly to the printer is only available via Alt-P.

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.l),d0
bd(An,Xn)	add.w $10000(a1,a2.l),d0 ('020 only)
([bd,An],Xn,od)	add.w ([10,a1],a2.l,20),d0 ('020 only)
([bd,An,Xn],od)	add.w ([10,a1,a2.l],20),d0 ('020 only)
(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.l),d0
bd(pc,Xn)	add.w $10000(pc,a2.l) ('020 only)
([bd,pc],Xn,od)	add.w ([10,pc],a2.l,20),d0 ('020 only)
([bd,pc,Xn],od)	add.w ([10,pc,a2.l],20),d0 ('020 only)

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 addressing modes marked 68020 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    #$1234567812345678,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

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 -d1 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 4 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 unprefixed 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.

-l(list) 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 68020 code is encountered. The -m must be immediately followed by one of the letters from the following list:

- **0** Used for 68000 target. Provides warning flags if you attempt to use 68020 only instructions. This is the default case.
- **2** Used for 68020 target. Turns off the warnings supplied in the -m0 option.
- **3** Used for 68030 target.

-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 68000, 68020, and 68030 instructions using the standard Motorola syntax. 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.

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 type and size combinations for relocation information on the 68000 are summarised in the following table:

Type	Size (bytes)	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.

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.

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      expression
IFEQ    expression
IFNE    expression
IFGT    expression
IFGE    expression
IFLT    expression
IFLE    expression
```

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     label
IFND    label
```

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     'string1','string2'
IFNC    'string1','string2'
```

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.

```
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.).

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 __stdargs 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. Structures, unions and double precision floating point numbers, 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
__d1        __a1
__d2        __a2
__d3
```

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 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 char 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
            move.l    6(sp),a0        changing stack offsets
@hash
            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                                   ; regargs 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--)
                    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

Branch out of range for 8-Bit offset

A short branch to a label outside the range of an 8-bit offset was specified. This can be cured by simply changing the branch size to word.

Branch out of range for 16-Bit offset

A word branch to a label outside the 16-bit range was specified.

Can't branch short to EXTERN

Asm does not allow short branches to an external label, causing this error.

Can't create object file

It was not possible to generate the object file. This can be caused by invalid -o options, disk full, protected files, etc.

Can't open source file

The source file could not be opened, often caused by an incorrect filename.

Combined output file name too large

The object file prefix specified from the command line caused the output filename to exceed the maximum length of 128 characters.

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.

Constant too large

An invalid ROM constant number was specified for an FMOVECR instruction.

Data generation must occur in reloc section

A data generation operation other than DS appeared in an OFFSET section.

Definition symbol not found

Internal error, should never happen.

Duplicate label

More than one definition of the same label was encountered.

Duplicate macro definition

A macro was defined more than once.

Duplicate section name

A section name was re-used illegally.

ELSE/ENDIF not found

An unterminated IFcc directive was encountered.

END directive assumed (W)

This warning notifies you that there was no explicit END directive in the source file being assembled.

Error writing object file Execution terminated

A DOS error occurred whilst writing the object file to disk.

Errors detected -- Processing terminated

This message appears at the end of the assembly process if any errors occurred. Any object file generated will be invalid.

External name table overflow

The maximum number of imported labels exceeded the maximum of 256.

External symbol defined

A definition for a label also declared as an external reference was encountered.

Extraneous data on input line (W)

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.

File name missing

The command line did not contain a file to assemble.

File not found

The file specified by an INCLUDE directive could not be found.

Generating 32 bit branch, code only valid on 68020

This warning is generated when assembling a long branch for the 68000 processor.

Generation argument count

Internal error, should never happen.

Illegal macro definition

The syntax of a macro definition was incorrect.

Immediate data size error
Immediate data too large

An arithmetic or logical operation was specified with an out of range immediate value.

Immediate mode not allowed

An instruction which does not support immediate addressing was encountered with an immediate mode operand.

Input line too large

A source line exceeded the maximum length of 255 characters.

Invalid Addressing Mode

Generated by an illegal addressing mode being supplied to certain 68020 or floating point instructions.

Invalid command line option

The assembler was invoked with an unrecognised command line option.

Invalid control section parameter

A CSECT directive with invalid parameters was encountered.

Invalid Data Size

The vector of a BKPT instruction was out of range.

Invalid destination mode

The second operand of an instruction was specified with an illegal addressing mode.

Invalid Effective Address for Opcode

One of the address translation cache family of instructions contained an illegal addressing mode.

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.

Invalid file name

The filename specified for an INCBIN or INCLUDE directive was not valid.

Invalid generation function index

The assembler attempted to generate an illegal instruction.

Invalid Length

A LINK instruction was encountered with an illegal stack offset.

Invalid list option

The assembler command line contained an unsupported listing option.

Invalid mode

An illegal addressing mode was used with an instruction.

Invalid opcode

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

Invalid operands for this opcode

This error can be caused by invalid addressing modes, data size, macro parameters etc.

Invalid origin

An assembly directive causing incorrect data alignment or origin was found.

Invalid relocation type/size combination

The specified relocation type and relocation data size specified in a CSECT directive are not available.

Invalid shift count

The bit count contained in a shift or rotate instruction was out of range.

Invalid Size
Invalid Size Field
Invalid Size For Mode

Each of these errors are caused by an invalid or unsupported data size extension to an instruction or addressing mode.

Invalid source

A MOVES instruction was specified with an invalid source operand.

Invalid source mode

The first operand of an instruction contained an unsupported effective address.

Invalid string

A define constant or condition directive contained an invalid string.

Invalid symbol

A symbol containing an illegal character or characters was declared.

Invalid vector

This error is generated if a TRAP instruction has an out of range vector.

-i option ignored

The maximum number of include directories which can be specified on the command line has been exceeded.

k-factor out of range

The k-factor specified for an FMOVE instruction on packed data was out of range of the legal values.

Label ignored (W)

The label before a directive, such as a conditional, is not a recognised syntax and has been ignored.

Label not found in pass 2

Internal error, should never happen.

Label offset different in pass 2

A phasing error caused by different code being generated on the first and second pass.

Lexical result overflow
Lexical type error
Lexical value overflow

Internal errors, should never happen.

Long Branches to EXTERNs not supported by the Linker

The Lattice linkable object file format does not support branches to external labels using a long-word offset.

Macro argument too large

A macro invocation was encountered with an argument string which was too long.

Macro buffer overflow

A macro definition was too long.

Macro nesting level exceeded

This error occurs when a macro definition references other macros too many levels deep.

Macro substitution line overflow

The substitution of macro arguments caused the line to overflow.

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.

Missing label

An EQU or SET directive was encountered with no corresponding label.

Missing macro definition

The definition of a macro could not be found.

Must occur inside section

A data generation directive was used outside a control section.

Not enough memory

The assembler ran out of memory when trying to allocate some internal buffer space.

Not inside macro definition

An assembly directive only valid within a macro definition, such as ENDM, was encountered outside a macro.

Not inside scope of IF directive

An ELSE directive was found which did not lie within a conditional control block.

Number Too Big

A value in an expression overflowed the allowable range.

Options beyond file name ignored

Any command line options specified *after*, rather than *before* the file to be assembled have been ignored.

Public symbol not defined (W)

The program source contained an XDEF directive of a symbol which was not defined in the program.

Seek error on object file Execution terminated

An attempt was made to move past the end of an object file. This is usually caused by an invalid RORG directive.

Target out of range

This error is generated if a DBRA instruction to a label which is out of range is found.

Too many control sections

This error signifies that the number of SECTION or CSECT directives has caused more than the allowed number of sections to be generated.

Unknown segment type

The type specifier for a SECTION or CSECT directive was other than CODE, DATA or BSS.

Unrecognized opcode

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

Value out of range for mode

An out of range value was used in an addressing mode.

Value out of range for PC Relative addressing

An out of range value was used in a PC relative addressing mode.

Warning 68020 or 68030 opcode generated (W)

This warning is generated when a non-68000 processor instruction is encountered and can be disabled from the command line by specifying an alternate processor.

Internal Errors

These are internal errors generated when the assembler encounters a situation which should not occur internally. If you encounter one of these please send us an example piece of source code which demonstrates the problem.

Memory freed incorrectly

Mode lexical pointer error

Section not found in pass 2

The Lattice C 5 Tools

Lattice C 5 comes with several tools which are non-essential to the operation of the compiler, but which can enrich the programming environment.

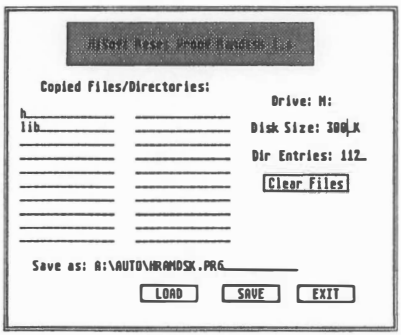
hramdsk — Reset Proof RAM Disk

If you have a megabyte or more of RAM then you can use some of this memory as a very fast disk which will make a hard disk seem slow. The problem with many ramdisks is that they disappear when you press reset. If you are developing a new GEM-based program and are having problems with your mouse or menu you can need to reset quite often.

RAMINST.PRG and RAMINST.RSC let you set up a ramdisk that will survive resets. Whilst this will work in 99% of cases ,occasionally, if a program crashes in a particularly nasty way you will lose the contents of the ramdisk. To avoid this save the source code on to a real disk before running your program.Hramdsk will additionally copy the files and folders that you would like on the ramdisk automatically when you switch on.

The version of HRAMDSK.PRG that we supply with Lattice C is configured for our recommend setup for users with one floppy disk and one megabyte of RAM as described in the Installation Guide. Before running RAMINST to tailor the ramdisk to your preferences, HRAMDSK.PRG should normally be in the AUTO folder on the current disk as this will be used as a basis for the ramdisk driver.

To run RAMINST copy it just double-click on RAMINST.PRG. If the driver cannot be found in the AUTO folder is not you will be presented with a file selector to enter the drive, path and file to load. Once the driver has been loaded, RAMINST will present you with a GEM dialog box like this:



You can change the files and folders that are copied by clicking on one of the relevant fields and editing it. To clear out the existing names click on the Clear Files button.

You can use simple filenames, filenames with wildcards or directory names. If you specify a directory name the entire directory (and any sub-directories within it) will be moved to the ramdisk. Note that you can specify the drive from which the files are copied.

Change the size of the ramdisk by simply modifying the Disk Size field. This is the amount of memory used by the disk and is slightly larger than the data size because some of the space is used for the directory and the file allocation table. You can use any size you like, subject to the available amount of RAM.

Normally you can have up to 112 directory entries in your main directory (just like a standard floppy disk) and this is usually sufficient. However you may change the maximum number of directory entries using the Dir entries field of the dialog box. For example, if you have a two and a half Megabyte ramdisk and using it to store (amongst other things) all the relocatable files for your 100 module wonder program and don't want to put them in a sub-directory, you can use this option to increase the number of directory entries.

If you don't like the ramdisk being called drive M you can change this too. You can also change the disk, directory and file to which the driver will be saved.

Finally if you wish to modify a differently installed version of the ramdisk you can click on the Load button and load another file.

Note that the ramdisk is only designed to be run from the AUTO folder; it can't be run once the system has booted.

lcompact

Header file compressor

This command is used to create a compressed version of a header file that may be processed more quickly by the compiler.

```
lcompact infile outfile
```

The lcompact command compresses a file by performing four basic operations:

Compression

Multiple and meaningless blank characters are removed. Expressions are analysed according to standard C precedence rules to determine which are safe to remove.

Elimination

All comments and unnecessary blank lines are removed.

Transformation

Certain sequences such as hex constants are converted into more efficient representations. For example, 0x0000001 is transformed to 1. These transformations take place only if the secondary representation is more space efficient.

Tokenisation

A limited number of common keywords are reduced to a single token character with the high order bit set. This fixed set of keywords is known to the compiler and will be automatically expanded as they are encountered.

The end result of compacting a header file is anywhere between 20% and 75% smaller depending upon the original contents. The compiler will automatically expand the header file on input so that error messages will print out the original line (without comments) for a diagnostic.

This utility program disassembles an object file produced by the Lattice C Compiler and produces an output listing consisting of assembly-language statements, possibly interspersed with the original C source code.

```
omd >output options object source
```

The object field is required and gives the complete object file name. That is, omd will not automatically supply the .o extension.

The source field is optional; if present, it must be the complete source file name. When this field is used, you should have compiled the source file with the -d option (see the lc command) to produce the debugging information in the object module that allows omd to associate a particular source line with the object code that was generated. If you did not use the -d option, then the C source lines will not appear in omd's output.

The >output field is optional and redirects omd's output from the screen to an output device or file. Most programmers use omd by redirecting its output to a file and then printing the file. See the example below.

The options field need not be present. The Atari implementation of omd only accepts the following option:

-x Override the default size of the buffer used to hold the external symbol section of the object module. For example, -x200 establishes a buffer that can hold 200 external symbols, which is the default. You should increase this value if omd reports that there are too many external symbols.

Example

This example compiles MYPROG.C to produce MYPROG.O, which is then disassembled. The disassembled listing is saved in the file MYPROG.LST.

```
lc -d myprog
omd >myprog.lst myprog.o myprog.c
```

The object module librarian oml can create a library file by combining object modules, generate a listing of the modules (and their public symbols) contained in a library, or manipulate modules within an existing library file.

Library files provide a convenient way of collecting object modules to be presented as a group of available components during linking; the linker then includes only those modules from the library which are actually needed by the program being built. Libraries are especially useful when several programs make use of common subroutines, since these subroutines can be placed in a library file and included, as required, when the programs are linked.

A library file is made up of object modules, each of which was originally a single file. Each module within the library is identified by a module name, which is normally obtained from the object module itself (the Lattice object module format defines a special *program unit* or module name record). This name is placed in the object module by the translator (assembler or compiler) program which generates it. Some modules may not contain a module name at all; in that case, the librarian assigns a module name of the form $nnn, where nnn is a decimal number indicating that the module was the nth un-named module encountered in the library.

In order to perform replacement of modules within a library file, it is necessary to ensure that the module contains a program unit or module name. The Lattice C 5 Compiler and assembler always place a module name in the object files they produce. The current versions of the compiler and assembler use the name of the object file. Thus, compiling ftoc.c produces an object module with the name ftoc.o.

When the linker examines a library file to find modules to be incorporated into a program, the module name is not important; instead, the linker decides if a module is needed on the basis of the public symbols it defines. A module may define one or more such symbols, which identify program components such as functions or variables. Because the presence of more than one definition for a symbol may cause confusion, the librarian warns when it examines or constructs a library file which includes multiple definitions of a symbol.

Each invocation of oml specifies a particular library file upon which operations are to be performed. Then, a sequence of one or more commands is used to indicate the desired operations.

Commands may be specified on the command line used to execute oml, or they may be read from stdin, or a combination of both. If no commands are specified on the command line, commands are automatically read from stdin, which is usually the user's console but can be redirected to a file. The special command @ (valid only on the command line) is used to switch command input from the command line to stdin; an explicit file name may be attached to the @ to force commands to be read from that file. Commands are read from a file or from stdin until an end of file condition is detected; if commands are being read from the user's console, a Ctrl-Z must be used to end command input.

Each command is specified as a single character, usually followed by a list of module names or object file names. Commands and file/module names are separated from each other by white space. If a command is followed by one or more names, the first name specified is *not* checked as a possible command; thus, names which might be confused with commands must be specified as the *first* name following a command.

The format of the command to invoke the librarian is:

```
oml [<cmdfile] [>listfile] [options] libfile [commands]
```

The various command line specifiers are shown in the order they must appear in the command. Optional specifiers are shown enclosed in brackets.

<cmdfile Causes commands to be read from the named file, provided that no commands are specified on the command line or that the @ command (see below) is not used to force commands to be read from stdin. If this option is omitted and neither of the above conditions is met, commands are read from the user's console.

>listfile Causes the listing output generated by oml to be written to the named file. If omitted, listing output is directed to the console.

libfile Specifies the name of the library file to be created or manipulated; this is the only command line field which *must* be present.

Options are specified as a minus sign followed by a string of characters which may not include white space. The options available are:

-b Forces the 'batch' mode of operation; in this mode no user interaction will occur.

-opfx Specifies that the output filenames for the x command are to be formed by prefacing the module name with pfx. Note that if a directory name is to be specified as a prefix, a trailing node separator (a backslash under GEMDOS) must be supplied on the prefix.

-s Causes a listing of the public symbols defined in the module to be included in the listing produced by the l command.

-v Forces the 'verbose' mode of operation; in this mode the librarian prints out its progress to date.

-x Causes a cross reference of symbols to be output in the listing produced by the l command.

Commands are specified by a single character; if alphabetic, either upper or lower case may be used. They are separated from each other and from elements of file or module name lists by any white space. The commands are:

@file Causes the remainder of command input to be read from stdin, or from the named file.

d list Deletes the named modules from the library. Since modules without program unit names are assigned module names by the librarian, it may be necessary to obtain a listing (via the l command) in order to determine the assigned $nnn name for the module which is to be deleted.

l Causes generation of a listing of the modules in the library after all other requested operations have been performed. If the -s option is used on the command line which invokes oml, the listing will include the public symbols defined in each module as well as a list of the module names themselves.

r list Replaces the named object files in the library, or adds them to the library if not already present. Note that replacement of existing modules in a library will work correctly only if the file name is the same as the module name. Note that a is a valid synonym for r.

x list Extracts the named modules from the library, creating files of the same names. Note that if the module name includes a path name, the librarian will attempt to create a file with that name. All files are created in the current directory unless the -o option is used; in that case, each module name is prefixed with the text specified on the -o option. If the special name * is specified in an extraction then oml will extract all files in the library. oml will terminate execution if an attempt to extract a module is unsuccessful. Note that it is an error to specify the same name in both a replacement and an extraction list.

If replacement modules or deletions are specified, a new version of the library file will be built, provided that no errors are detected. This new version is created first as a temporary file; when it has been completely built, the original library file (if it existed) is deleted, and the temporary file renamed. This sequence ensures that the original library file will not be affected if an error is detected.

Modules are always included in a library in topologically-sorted order, so that no backward references occur (except in the case of modules which reference each other, which are retained in the same order in which they are encountered).

Warning messages are produced if a module named in a deletion or extraction list was not found in the library or if a second definition for a public symbol is encountered in one of the modules to be included.

Examples

The following examples illustrate the use of oml. Remember that replacing modules within a library file only works correctly if the module name is the same as the file name.

Building a New Library

Create a list of the file names of the object modules which will make up the library. Then create the library using the following command:

```
oml new.lib r @name.lst
```

where new.llb is the name of the library to be created, and name.lst contains a list of the files to be included in the library. Note that creation of a new library is one occasion where the correspondence between file and module names is not required.

Extracting Modules from a Library

Use the following command to break out all of the modules from a library:

```
oml -o\object\ cfuncs.lib x
```

Note that this command will be successful only if no module names in the library cfuncs.llb contain path names. A file for each of the modules in the library will be created in the directory \object\ in this example.

Deleting Modules from a Library

Use the following command to delete modules from a library:

```
oml cfuncs.lib d tribe.o
```

This example deletes the module tribe.o from the library cfuncs.llb.

Listing the Modules in a Library

Use the following command to obtain a listing of the modules and symbols in a library file:

```
oml -s test.lib l
```

The listing may be saved to a file using I/O redirection:

```
oml >test.lst -s test.lib l
```

Strip is a utility for removing the symbol table and any symbolic debugging information, from an executable file.

```
strip file1 [file 2 ...]
```

Any number of files, which should include any extension, may be specified. If a file is not executable it is simply ignored.

Example

This example compiles MYPROG.C to produce MYPROG.O, which is then linked and then has its symbols and debug information removed:

```
lc -d3 -La myprog
strip myprog.ttp
```

wconvert

Resource Name Converter

Wconvert is a utility for converting the name files from the Digital Research and Kuma Resource Construction Sets. It is provided so that you can edit resource files produced using these programs with WERCS while retaining your names for trees and objects.

There are two versions of this program:

wconvert.prg lets you select the file to convert using the file selector. After it has converted one file it will let you select another one if you wish.

The second version is wconvert.ttp which takes the names of the file(s) to convert on the command line for CLI users. Wildcards may be used with this version.

Wconvert treats files differently depending on their extension:

DEF it is assumed to be a Digital Research RCS1 file.

RSD it is assumed to be a Kuma K-RSC file (actually the same basic format as RCS1).

DFN it is assumed to be a Digital Research RCS 2 file.

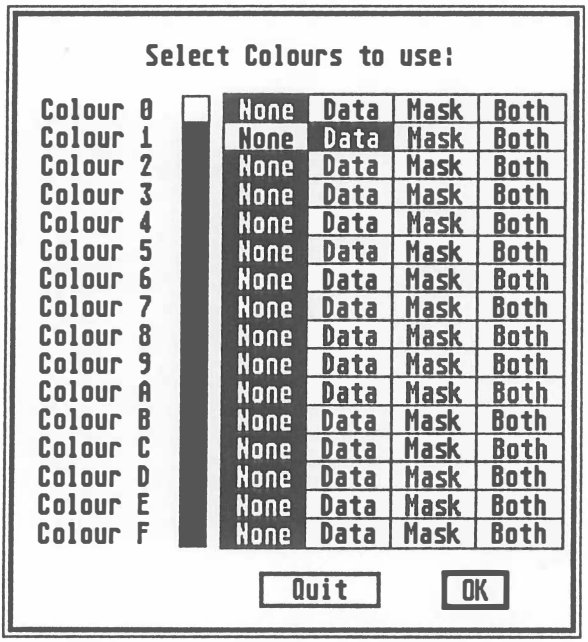
The file will be converted into a .HRD file of the same name and in the same directory as the old file, ready for use with WERCS. Remember to make sure that the Language and Case settings are correct when you edit the file with WERCS for the first time.

Wimage is a utility for converting parts of Neochrome and DEGAS format files into resource files.

After starting wimage and you will be prompted to enter a file to convert via a File Selector. This file may be a Neochrome format file (normally with an extension of .NEO) or un-compressed Degas/Degas Elite file (normally .PI3, .PI2 or .PI1). Wimage knows about medium and high resolution Neochrome format files even though Neochrome itself does not.

Converting colour pictures to Images and Icons has the disadvantage that GEM Icon and Images have only two colours. Also note that the maximum size of Images and Icons that can be converted is 128x128 pixels.

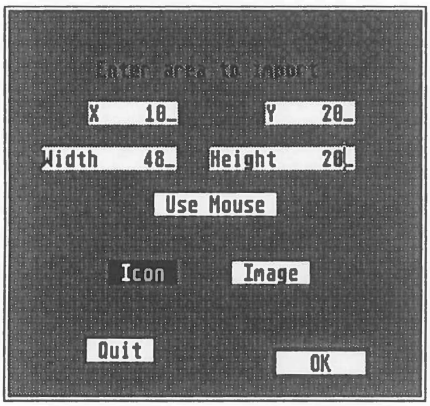
After entering the file name, the Image file will be loaded and you will be presented with a dialog box like this:



The box above is for a low resolution picture. If the picture is a medium or high resolution picture then only the appropriate colours will be displayed. If you are converting from a low resolution picture and the screen is in low resolution mode then the boxes after Colour 2 etc. will be in the appropriate colours as displayed by Neochrome.

This dialog box enables you to indicate which colours are to be treated as Data and Mask bits in the Icon or Image that wImage produces. If Both is selected for a particular colour then the corresponding pixels of this colour will be set in *both* the Data and Mask. If Data is selected they will be *set* in the Data but *reset* in the Mask. If Mask is selected then pixels of this colour will be *set* in the Mask but *reset* in the Data. If None is chosen then the bits will be reset in *both* the Data and the Mask. When importing an Image, only the Data setting is used.

You will then be presented with a dialog box like this:



Enter the area of the picture that will be converted. Indicate whether you are producing an Icon or an Image by clicking on the appropriate radio button.

If you wish to use the mouse to select the area, ensure that the Use Mouse button is selected; this button will be disabled if this picture may not be displayed in the current resolution. If Use Mouse has not been selected then the area to import is taken from the co-ordinates entered.

If you click on OK you will then be prompted to use a File Selector to enter the output file to which your Image or Icon will be written in the form of a resource file. This can be imported into another resource file using the Import Image item from the WERCS File menu.

After the file has been saved you will be given the opportunity to import another image or to quit the WImage program.

Appendix A Implementation Behaviour

This appendix tries to detail the areas Lattice C 5 which are traditionally different across compilers, and often left undefined. Reliance on any of this information will, in general produce non-portable programs; note that this *includes* reduced portability to other Lattice C systems.

This section is based around Appendix F.3 of the ANSI C Standard document and the paragraph numbers relate to those paragraphs in that standard.

Translation

2.1.1.3

Diagnostic messages are issued on the console device describing the error situation encountered. For syntax violations the compilation is subsequently terminated.

Environment

2.1.2.2.1

The arguments to main() are parsed from the command line as passed to the program via the GEMDOS Pexec() call. Whitespace characters are considered to be separators unless enclosed in single or double quotes. The argv(0) string points to an empty string as the program name is not available.

When started by a process supporting the Atari extended command line format, the arguments to main() are as supplied by the parent process.

A third argument is passed to main() representing the environment variable vector, as described under environ.

2.1.2.3

An interactive device is assumed to be those for which GEMDOS indicates that Fseek() cannot be performed.

Identifiers

3.1.2

The number of significant characters is specified by the -n compile time option. This has a default of 31, and a maximum of 100.

The linker treats all characters as significant, with full case significance.

Characters

2.2.1

The source and execution character sets are the Atari ASCII set.

2.2.1.2

No shift states are used for encoding of multibyte characters.

2.2.4.2.1

There are always eight bits in a char giving a range from -128 to 127 for signed characters or 0 to 255 for unsigned.

3.1.3.4

The source character set is mapped as-is to the execution character set

Multiple character constants are supported when using the -cm flag. The lexical characters are parsed right to left and are packed low byte upwards. Hence the character constant 'ALEX' would pack:

The "C" locale is used to convert multibyte characters into corresponding wide characters (codes) for a wide character constant.

3.2.1.1

The type char defaults to signed, but may be configured as unsigned at compile time (-cu).

Integers

3.1.2.5

All integer types are two's complement.

Type int is configurable to be 16 or 32 bits. The default setting is 32 bit, with 16 bit being available when using the -w option.

Type short is 16 bits; type long is 32 bits.

3.2.1.2

The result of converting an integer to a shorter signed integer, or the result of converting an unsigned integer to a signed integer of equal length is truncation to the lower bits of the assigned integer.

3.3

Bitwise operations on signed integers produce results as if the integers were unsigned.

3.3.5

The sign of the remainder on integer division is the same as that of the quotient.

3.3.7

Right shift of a negative-valued signed integral type produces a sign extended type.

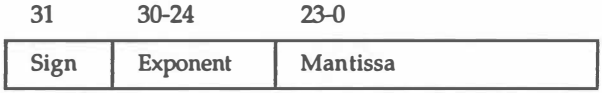
Floating Point

3.1.2.5

The floating point format used is the IEEE standard format for both float and double. long double is implemented as double in the current release.

Single-precision Floating Point

The single precision IEEE format represents a number in 4 bytes. Note that all calculation is performed using doubles so that use of floats will only reduce storage space and not increase the speed. The bit layout is:



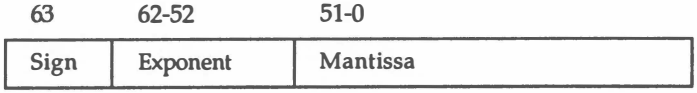
The sign bit is 0 for positive numbers and 1 for negative numbers. The mantissa has an implied binary point at bit 23 and thus ranges in value from 1.0 to <2.0. The exponent is held in excess 127 format. The IEEE denormalised format is not currently supported. When the exponent is 255, the value represents Not-A-Number (NaN), the type of which is determined by the mantissa. Zero mantissas indicate Infinity (∞), whilst non-zero mantissas indicate other NaN conditions. The number zero is represented by all bits zero.

The following (non-portable) definition may be used to access the individual fields of a float atomically:

```
union
{
  float f;
  struct
  {
    int s:1;  /* sign */
    int e:8;  /* exponent */
    int f:23; /* mantissa */
  } b;
};
```

Double-precision Floating Point

The double precision IEEE format represents a number in 8 bytes. The bit layout is:



The sign bit is 0 for positive numbers and 1 for negative numbers. The mantissa has an implied binary point at bit 51 and thus ranges in value from 1.0 to <2.0. The exponent is held in excess 1023 format. The IEEE denormalised format is not currently supported. When the exponent is 2047, the value represents Not-A-Number (NaN), the type of which is determined by the mantissa. Zero mantissas indicate Infinity (∞), whilst non-zero mantissas indicate other NaN conditions. The number zero is represented by all bits zero.

The following (non-portable) definition may be used to access the individual fields of a double atomically:

```
union
{
  double d;
  struct
  {
    int s:1;  /* sign */
    int e:11; /* exponent */
    int f1:20;  /* high bits of mantissa */
    int f2:32;  /* low bits of mantissa */
  } b;
};
```

3.2.1.3

In default mode (-fl) truncation of an integer to floating point is towards zero. In -f8 mode the direction of truncation is as specified by the maths coprocessor (user-selectable).

3.2.1.4

In default mode (-fl) truncation of an floating point number to a narrower floating point type is towards nearest. In -f8 mode the direction of truncation is as specified by the maths coprocessor (user-selectable).

Arrays and Pointers

3.3.3.4, 4.1.1

size_t is *always* of type unsigned long int. Traditionally it has been unsigned int.

3.3.4

Casting a pointer to an integer requires that the target type be long (or int in default long integer mode), or truncation will occur. Casting an integer to a long is as if the integer were first extended to long.

3.3.6, 4.1.1

ptrdiff_t is of type long.

Registers

3.5.1

The compiler honours as many register variable declarations as possible on a lexically first encountered basis for integral, pointer and floating point types (when the -f8 option is used). If the global optimiser is used all registers are remapped to provide maximal usage.

The number of registers available for register declarations is not fixed and hence is undefined. Typically a minimum of 4 scalar and 2 pointer variables will be available.

Structs, Unions, Enums, and Bit-fields

3.3.2.3

Accessing a member of a union object using another member of a different type produces undefined behaviour.

3.5.2.1

The members of a structure are aligned according to the alignment restrictions of the basic type. Hence the structure:

```
struct
{
  char x;
  char y[3];
};
```

would *not* align y. Whereas the structure:

```
struct
{
  char x;
  int y[3];
};
```

would align y. The amount of padding inserted (when required) is determined by the -I flag which will force long word alignment. When -I is not present the default alignment is word alignment.

A plain Int bit-field is treated as an unsigned Int bit-field

The members of a bit-field are always elements of a long. This means that the members are also of type long.

A bit-field *never* straddles a long boundary.

A bit-field is packed from the top down hence the definition:

```
struct
{
  unsigned abcde:5;
  unsigned fghijk:6;
  unsigned lmn:3
  unsigned opqrs:5;
  unsigned tuvwxyz:7;
};
```

would pack as:

31-27	26-21	20-18	17-13	12-6	5-0
abcde	fghijk	lmn	opqrs	tuvwxyz	??????

3.5.2.2

Enumeration types have the integral type int.

Qualifiers

3.5.3

Whether a reference to an object with volatile qualified type between sequence points constitutes an access is undefined.

Declarators

3.5.4

The maximum number of declarators that may modify an arithmetic, structure, or union type is limited by available memory.

Statements

3.6.4.2

The maximum number of case values in a switch statement is limited by available memory.

Preprocessing Directives

3.8.1

The value of a single-character character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set. Such a character constant may have a negative value.

3.8.2

System include files (those in < > brackets) are located initially in the current directory, then in the directory mentioned in the INCLUDE environment variable, then in directories mentioned on the command line via the -I option.

User include files (those in "'s) are located initially in the current, then in directories mentioned on the command line via the -I option, finally in the directory mentioned in the INCLUDE environment variable.

The mapping of includable file names to external source names is obtained by taking the last eight characters of any 'directory' portions (\ or / delimited) together the same portion of the final filename and any extension.

3.8.6

The behaviour on the #pragma Inline directive is as described in the section **LC, The Compiler**. This is the only supported #pragma directive.

3.8.8

The definitions for __DATE__ and __TIME__ when respectively, the date and time of translation are not available are the date and time of compiler build.

Appendix B
Resource Details

This appendix contains detailed information on programming with resource files, compiling the WERCS example program and the file formats used by WERCS.

This section of the manual is designed to help you when programming with resource files that have been created by WERCS. The first section describes the objects and their attributes, whilst the second considers the programming aspects of these data structures. It continues by considering the multi-language support available from WERCS and concludes by discussing the WERCS' specific file formats.

Objects

There are thirteen different types of objects as follows:

Box

A Box is a rectangle whose interior colour and fill pattern is controllable, as is its border thickness.

BoxChar

A BoxChar is a graphic box containing a single text character. It also has colour and border size attributes.

BoxText

Similar to Text objects (see later) but in addition surrounded by a border whose size and colour can be specified.

Button

A Button is displayed as a centred string of characters with a border. If the *default* flag is set, pressing the Return key in the standard form-handler is the same as clicking on the Button. A Default button is shown with a wider border. A new Button created with WERCS has its Selectable and Exit flags set.

FBoxText

An FBoxText object is a formatted BoxText object; in addition to the normal Text attributes, it also has border attributes, a template and two extra strings for text entry. These extra strings are called the Template and Valid fields. The AES displays the Template string as if it were displaying any other type of text except that for every underline character it displays a character from the main text string. For example, a date field object might consist of a Text entry of:

```
011088
```

and a Template of

```
Date:__/__/__
```

this would be displayed as

```
Date:01/10/88
```

if it were an FBoxText.

Remember: underline characters are entered as tildes (~).

The Valid string is used when the object is used as a Form using the GEM form_do command. The Valid string specifies which characters may be typed for each underline character in the Template string. The different validation characters are:

X	all characters allowed
9	Only 0-9
A	A-Z and space
a	A-Z, a-z, 128-255 and space
N	A-Z, 0-9 and space
n	A-Z, a-z, 0-9, 128-255 and space
P	A-Z, a-z, 0-9, 128-255, \ : ? * . _
p	A-Z, a-z, 0-9, 128-255, \ : _
F	A-Z, a-z, 0-9, 128-255, : ? * _
f	A-Z, a-z, 0-9, 128-255, _

In the above A-Z includes non-English capital letters. 128-255 means that all characters greater with value greater than 128 can be used, including lower case non-English letters and the £ sign.

You can use different validation characters in the same string if you wish. Thus for the date example above we would use the 9 character for all 6 character positions since the only characters allowed in dates are digits.

The most commonly used of these validation digits are probably X and 9. Note that if you wish to enter negative numbers you have to use X (otherwise the - sign would not be allowed).

Also the pathname options (P and p) are of limited value as a number of software producers sadly use illegal pathname characters such as - in their filenames. All but the X, F and f validation characters also have the undesirable feature with the first, 1.0, operating system ROMs of crashing the system when you press _!

Note that whilst validation characters, P, p, F and f allow you to enter lower case letters these are echoed as the upper case equivalents.

If otherwise illegal characters are present in the Template string then the AES will skip past them if they are entered. With our date example typing / will skip to the next field even though / is not otherwise a valid character.

You should ensure that there are at least as many underlines in the Template string as there are characters in the main Text string; otherwise all of the latter will not be displayed. If you are intending to use this object as a Form *in situ* as normal, you should have the same number of characters in the Text string as there are underlines in the Template; if you don't observe this then if the user types a long string, the next string from your resource file will be corrupted. This restriction does not apply if you are intending to change the address of the Text field when the resource file is loaded.

So, in general, ensure that there are exactly the same number of characters in your Text string as there are underlines in your Template string.

Surprisingly, the Valid string does not have to contain a character for every underline in the Template string; if all the validation characters are the same then you can use just one. We have not seen this officially documented but it certainly works on all versions of the operating system we have used and can lead to considerably reduced resource file and memory usage if you have long strings.

If the first character in a text field is the at-sign (@), then form_do will display your string as underlines and place the cursor at the start of the string. Thus you can enter a blank string of n characters by typing, say ~, n-1 times, press cursor left until you are at the start of the string and then press @. The string will then disappear; but don't worry; it will be stored in your file ready for use.

FText

Similar to a FBoxText (see above) but without border attributes.

IBox

An IBox is a so-called *invisible box*, similar to a Box but hollow. It is only truly invisible if its border has a thickness of zero.

Icon

This consists of two bit-map images, one for data and one for a mask. In addition a string of characters and a single character are also associated with it. Icons also have their own foreground and background colours.

Image

An Image is a graphic bit-map with a foreground colour attribute. It differs considerably from an Icon; it has no mask (so cannot be distinguished on a patterned background), and no associated text or single character.

ProgDef

This type of object is for advanced programmers only. It allows you to create your own types of object by supplying your own drawing routines. ProgDefs are displayed in WERCS as boxes with a diagonal line. ProgDefs are also known as *UserDefs*. We have seen the latter term used mainly in older documentation; a hangover perhaps from days when a Digital Research *user* was someone who wrote the assembly language to install CP/M on their computer.

String

A String is a sequence of characters drawn in black and in the standard system font. If you require different sized or coloured text you should use one of the formatted text object types.

Text

Actually graphic text; this is a sequence of characters that can be displayed in the system font or in a small font and can be left-, centre- or right-justified.

Title

Objects of type Title are only used as Menu titles. Their use in other types of tree is not recommended; they have the same attributes as Strings.

Flag Types

The different flag types for objects are as follows:

Selectable

The Selectable flag is used in conjunction with the form_do AES routine. If this flag is set then if the user clicks on the object during a form_do call it will be highlighted and the Selected state bit will be set. If the Selected bit was already set, the object will be shown as normal and the Selected bit reset.

Thus setting this bit effectively turns any object into a Button without changing the appearance of the object. All Buttons that are to be used as such should have this bit set; this is the default when you create a new Button with WERCS.

Default

The Default bit tells the AES that this is the default button of the form, i.e. the one which will be returned if the user types Return.

Normally this is used for Buttons but can also be used for other types. With Buttons the Default bit causes the object to be displayed with a wider border so that the user can see the default. For other objects there is no change in the screen display.

If the Default bit is set for an object you should normally also set the Selectable and Exit bits.

We do not recommend having more than one Default item in a form; the user, your program and the AES are likely to get confused.

Exit

The Exit bit is used to indicate that clicking on this object will cause form_do to return to your program, with the index of this object as its result. If the Exit bit is not set the user can continue to edit the Form.

This bit can be used for any type of object, but only with Buttons is the size of the border increased to indicate to the user that this is an Exit Button. When you create a new Button using the Object menu this bit will be set.

The Selected bit should be set whenever the Exit bit is set.

Editable

The Editable bit should only be set for the FText and FBoxText objects; this indicates that the user may edit the text in this field. If you set this bit for other types of objects the AES will mis-behave often causing the system to bomb. The fields in the TEDINFO structure used by the AES for FText and FBoxText objects must conform to strict rules as described under FBoxText.

There is no need to set other flags in conjunction with the Editable flag.

Do not use an Editable text field as the *last* object in a Form; all the current versions of the operating system will crash if you press cursor down when editing this field.

Radio Button

The Radio Button bit is used to indicate that an object is one of a set of radio buttons. The objects need not be of type Button.

Every sibling of the object should have the Radio Button bit set; to ensure this you can use an IBox to surround just the objects that you wish to be Radio Buttons. Radio Button objects should have the Selectable flag set.

For an example of programming with Radio Buttons see the WTEST program.

Touch Exit

The Touch Exit bit is used to tell the AES to exit form_do when the user moves the mouse pointer over an item and clicks on it. The exit occurs when the mouse button is pressed down (rather than released as in the case of the Exit flag). Touch Exit also differs from Exit in that the button need not be Selectable. When using form_do, if the user double-clicks on a Touch Exit object then the top bit of the return value will be set. Even if your program is not interested in double-clicks, you still need to mask off the top bit.

This flag may be used with any kind of object.

Hide

The Hide bit is used to hide an object and all its children from the AES. This means that the object is not displayed by objc_draw and will not be found by objc_find. This is useful when you wish to remove part of a tree temporarily, without re-organising that tree. For example, WERCS itself uses this facility when drawing the Extras dialog box to ensure that only appropriate types of objects are shown.

If you have hidden an object using WERCS you cannot use the Hide command from the Flags menu to unhide it again because you cannot select it; instead select its parent and then use the UnHide Children command from the same menu instead.

Flag States

The flag states for objects are as follows.

Selected

If the Selected flag is set, it indicates that the object will be displayed highlighted. This bit is changed from 0 to 1 or from 1 to 0 if the object is Selectable when the user clicks on the appropriate object. Any type of object may have this bit set.

Crossed

The Crossed bit causes the AES to draw a white diagonal cross through the object. If the object is Selected then the cross is displayed as black. This flag can be used on all objects except IBoxes.

Checked

If the checked flag is set the AES will draw the object with a black tick mark, ✓, inside it with the tick in the top left corner. When the object is Selected the tick is shown in Black. The Checked flag may be used for any type of object including IBoxes.

Disabled

If the Disabled flag is set for an object then it is shown greyed, that is, with less intense colour than normal. In addition, Disabled objects may not be Selected when using form_do or as part of a Menu even if they have the Selectable bit set. Note, though, that Disabled Editable fields may be edited!

Outlined

If the Outlined bit is set then the object is drawn with a black box outside it. Note that this does not form part of the object as far as objc_find, for example, is concerned. This bit may be used with all types of objects.

Shadowed

If the Shadowed bit is set for an object a shadow is drawn outside the object in the object's border colour; this includes Buttons. The Shadowed bit has no effect on objects without a border.

Selecting both Outlined and Shadowed attributes produces a messy display of the object and should be avoided.

Object, Flags and States Summary

The following table shows which attributes change the appearance on screen for each type of object. Text Attr refers to the alignment and size of text:

	Fill Pattern	Fill Colour	Xparent /Opaque	Border Colour	Border Size	Text Colour	Text Attr
Box	✓	✓		✓	✓		
BoxChar	✓	✓		✓	✓	✓	
BoxText	✓	✓	✓	✓	✓	✓	✓
Button							
FBoxText	✓	✓	✓	✓	✓	✓	✓
FText			✓		✓	✓	✓
IBox				✓	✓		
Icon		✓				✓	
Image						✓	
ProgDef							
String							
Text			✓		✓	✓	✓
Title							

The following table shows the effect of particular flag/state sets for a number of the Flags. Remember that Selectable, Radio Button, TouchExit, Selected and Outlined may be used for all types of objects except Titles:

	Default	Exit	Editable	Crossed	Disabled	Shadowed
Box	☞	☞	✹	✓	☞	✓
BoxChar	☞	☞	✹	✓	✓	✓
BoxText	☞	☞	✹	✓	✓	✓
Button	✓	✓	✹	✓	✓	✓
FBoxText	☞	☞	☞	✓	✓	✓
FText	☞	☞	☞	✓	✓	✖
IBox	☞	☞	✹	✖	☞	✓
Icon	☞	☞	✹	✓	✓	✖
Image	☞	☞	✹	✓	✓	✖
ProgDef	☞	☞	✹	✓	✓	✖
String	☞	☞	✹	✓	✓	✖
Text	☞	☞	✹	✓	✓	✖
Title	☞	☞	✹	✓	✓	✖

Key:

- ✓ Changes appearance of object and the behaviour of the AES.
- ☞ Changes the behaviour of the AES but not the appearance.
- ✖ Has no effect.
- ✹ Causes the machine to crash with bombs.

Programming with Resources

This section details the various data structures and object types, together with common AES programming algorithms. This section uses the standard names and typedefs for the data structures and their components which are supplied in the header file aes.h.

Tree Structure

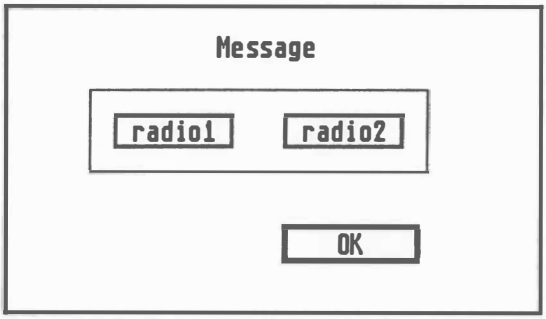
OBJECT Structure

A tree is stored in memory as an array of objects. Each object has pointers to allow the AES to tree-walk as required. The structure is as follows:

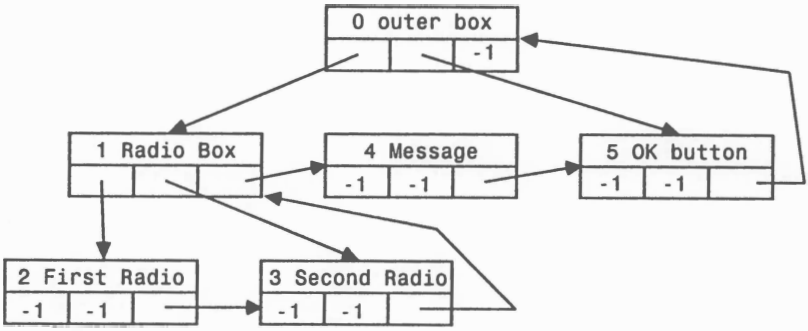
```
typedef struct object
{
  short ob_next;              index of object's next sibling
  short ob_head;              index of first child or -1 if
                              none
  short ob_tail;              index of last child or -1 if
                              none
  unsigned short ob_type;     object type (high byte is
                              ignored by the AES and used for
                              extended object numbers)
  unsigned short ob_flags;
  unsigned short ob_state;
  void *ob_spec;              depends on object type
  short ob_x;                 X co-ordinate of object
                              relative to parent (in pixels)
  short ob_y;                 Y co-ordinate of object
                              relative to parent (in pixels)
  short ob_width;             width of the object in pixels
  short ob_height;            height of the object in pixels
} OBJECT;
```

All the fields are present for all objects although the ob_spec field depends on the object type and is usually a pointer to another structure as described below.

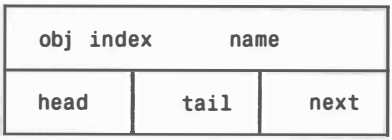
When it is loaded into memory an object tree is like an array of records. The first object (with index 0) is called the *root object*. It is normally the outer Box of a dialog box. Each object in the tree has three fields called ob_head, ob_tail and ob_next. These hold integer values that dictate to the AES the structure of the tree. Fortunately you do not normally need to access these directly, WERCS does it for you. As an example, say we have a dialog box like this:



The tree structure this represents can be shown as:



where each box represents:



When this is stored in memory the index and first three fields of the various objects will be:

Index	ob_head	ob_tail	ob_next	name
0	1	5	-1	outer box
1	2	3	4	Radio Box
2	-1	-1	3	First Radio
3	-1	-1	1	Second Radio
4	-1	-1	5	Message
5	-1	-1	0	OK button

Object number 0 is called the *root* of the tree. Its *children* are Message, Radio Box and OK button. Radio box's *parent* is outer box; its children are First Radio and Second Radio; its *siblings* are Message and OK button. First Radio and Second Radio are *childless* and they are *grand children* of the root object, outer box.

Normally what is important with object trees is the tree structure not the order that the items are in memory. For example, don't assume that the first child immediately follows its parent.

We can now give concise definitions of the ob_head, ob_tail and ob_next fields:

ob_head — points to the first child (or is -1 if childless)

ob_tail — points to the last child (or is -1 if childless)

ob_next — points to the next sibling or, if there are no more siblings, to the object's parent.

Values for the ob_type field, together with the interpretation of the ob_spec field will now be described, followed by the definition of the data structures to which they refer.

Box G_BOX 20

The ob_spec field contains the colour word (low word) and border thickness (high word).

Text G_TEXT 21

The ob_spec field contains a pointer to a TEDINFO structure; the te_ptext pointer within the structure points to the actual displayed text.

BoxText G_BOXTEXT 22

The ob_spec field contains a pointer to a TEDINFO structure; the te_ptext pointer within the structure points to the actual displayed text.

Image G_IMAGE 23

The ob_spec field contains a pointer to a BITBLT structure.

ProgDef G_PROGDEF 24

The ob_spec field contains a pointer to an APPLBLK structure.

IBox G_IBOX 25

The ob_spec field contains a colour word (low word, only border colour attribute used) and border thickness (high word).

Button G_BUTTON 26

The ob_spec field contains a pointer to the displayed string.

BoxChar G_BOXCHAR 27

The ob_spec field is used for the following fields:

bits 24-31	ASCII value of displayed character
bits 16-23	border size
bits 0-15	colour word

String G_STRING 28

The ob_spec field contains a pointer to the displayed string.

FText G_FTEXT 29

The ob_spec field contains a pointer to a TEDINFO structure. The text pointed to by te_ptext is merged with the template pointed to by te_ptmplt before display. The fill attributes in te_color are ignored.

FBoxText G_FBOXTEXT 30

The ob_spec field contains a pointer to a TEDINFO structure. The text pointed to by te_ptext is merged with the template pointed to by te_ptmplt before display.

Icon G_ICON 31

The ob_spec field contains a pointer to an ICONBLK structure.

Title G_TITLE 32

The ob_spec field contains a pointer to the displayed string.

Object Flags

The various flags in the ob_flags field have the following values as bits and as a hexadecimal mask:

Name on Menu	Standard Name	Bit	Mask
Selectable	SELECTABLE	0	0x1
Default	DEFAULT	1	0x2
Exit	EXIT	2	0x4
Editable	EDITABLE	3	0x8
Radio Button	RBUTTON	4	0x10
	LASTOB	5	0x20
Touch Exit	TOUCHEXIT	6	0x40
Hide	HIDETREE	7	0x80
	INDIRECT	8	0x100

The LASTOB bit is used by the AES to find the last object in an object tree; it is set for the last object and the last object alone. This bit is handled by WERCS for you but you may find it useful to access it if you write routines to manipulate trees in memory.

If the INDIRECT bit is set, the ob_spec field is treated as a *pointer* to the ob_spec field rather than the value itself. WERCS does not allow you to set this bit; if you need it then your program should set it and re-initialise the ob_spec field as required.

Object States

The following table gives the values as bits and masks of the ob_state field.

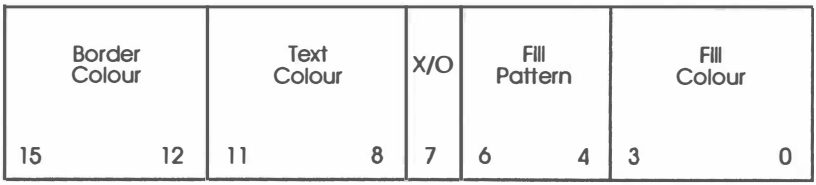
Name on Menu	Standard Name	Bit	Mask
Selected	SELECTED	0	0x1
Crossed	CROSSED	1	0x2
Checked	CHECKED	2	0x4
Disabled	DISABLED	3	0x8
Outlined	OUTLINED	4	0x10
Shadowed	SHADOWED	5	0x20

Border Thickness

The low byte of the high word in some ob_spec fields stores the border thickness in pixels. A value of 0 means no border, positive values give a border inside the object, negative values force it outside the object.

Colour Word

The colour word used in some ob_spec fields consists of the following components:



In the above diagram the numbers indicate the bits, so that the Border Colour is in bits 15-12, the four most significant bits of the first byte.

X/O is the Transparent/Opaque bit; Opaque is indicated by the bit being set.

Fill Pattern is as on the Fill menu with 0 indicating hollow- and 7 solid- fill.

The Border, Text and Fill Colours are as on the appropriate menus. The standard names for the colours are:

Colour	Value	Colour	Value
WHITE	0	LWHITE	8
BLACK	1	LBLACK	9
RED	2	LRED	10
GREEN	3	LGREEN	11
BLUE	4	LBLUE	12
CYAN	5	LCYAN	13
YELLOW	6	LYELLOW	14
MAGENTA	7	LMAGENTA	15

The L in the above names indicates *light*. If you *must* encode a colour word into your program the best base to use is hexadecimal.

TEDINFO Structure

This structure is used by the object types BoxText, FBoxText, FText and Text:

```
typedef struct text_edinfo
{
  char *te_ptext;       pointer to actual text
  char *te_ptmplt;      pointer to template; editable
                        portion denoted by underscores
  char *te_pvalid;      pointer to string containing
                        validation characters
  short te_font;        font used: 3=system font, 5=small
                        font
  short te_junk1;       reserved for future use
  short te_just;        text justification required:
                        0=left, 1=right, 2=centre
  short te_color;       object colour and pattern of box-
                        type objects (see previously for
                        word format)
  short te_junk2;       reserved for future use
  short te_thickness;   border thickness
```

```
  short te_txtlen;      length of te_ptext string
                        (including null)
  short te_tmplen;      length of te_ptmplt string
                        (including null)
} TEDINFO;
```

ICONBLK Structure

This is used by the Icon object type only:

```
typedef struct icon_block
{
  short *ib_pmask;      pointer to icon mask
  short *ib_pdata;      pointer to icon data
  char *ib_ptext;       pointer to the text displayed with
                        the icon
  short ib_char;        low byte is the displayed character,
                        high byte defines colour used - top
                        nibble is foreground colour, bottom
                        nibble is background
  short ib_xchar;       X co-ordinate of ib_char relative to
                        ib_xicon
  short ib_ychar;       Y co-ordinate of ib_char relative to
                        ib_yicon
  short ib_xicon;       X co-ordinate of icon relative to the
                        ob_x of the object
  short ib_yicon;       Y co-ordinate of icon relative to the
                        ob_y of the object
  short ib_wicon;       width of the icon image in pixels
                        (must be a multiple of 16)
  short ib_hicon;       height of icon image in pixels
  short ib_xtext;       X co-ordinate of icon's text relative
                        to the ob_x of the object
  short ib_ytext;       Y co-ordinate of icon's text relative
                        to the ob_y of the object
  short ib_wtext;       width of rectangle to display icon's
                        text in (centred)
  short ib_htext;       height of icon's text
} ICONBLK;
```

The bit images for the mask and data are stored as arrays of words.

BITBLK Structure

This is used by the Image object type and Free Images only:

```
typedef struct bit_block
{
  short *bi_pdata;   pointer to bit image
```

```
  short bi_wb;        width of image data in bytes (must be
                      even)
  short bi_hl;        height of image in pixels
  short bi_x;         source X co-ordinate
  short bi_y;         source Y co-ordinate
  short bi_color;     colour word (see previously)
} BITBLK;
```

bl_x and bl_y are used as offsets into the bit image given by bl_pdata; any bits before this will be ignored.

APPLBLK Structure

This is used by ProgDefs:

```
typedef struct appl_blk
{
  int (*ab_code)(PARMBLK *);     pointer to code to draw the
                                 object
  long ab_parm; passed as a parameter to the low-level
                                 drawing routine
} APPLBLK;
```

PARMBLK Structure

This is passed to ProgDef drawing routines:

```
typedef struct parm_blk
{
  OBJECT *pb_tree;       pointer to start of object tree
  short pb_obj;          the object index
  short pb_prevstate;    the old state of the object to be
                         changed
  short pb_currstate;    the new (changed) state of the
                         object
  short pb_x;            the pixel X screen co-ordinate of
                         the object
  short pb_y;            the pixel Y screen co-ordinate of
                         the object
  short pb_w;            the pixel width the object
  short pb_h;            the pixel height of the object
  short pb_xc;           the pixel X co-ordinate of the
                         current clip rectangle
  short pb_yc;           the pixel Y screen co-ordinate of
                         the current clip rectangle
  short pb_wc;           the pixel width of the current clip
                         rectangle
  short pb_hc;           the pixel height of the current
                         clip rectangle
```

```
    long pb_parm;          copied from the ab_parm value in
                           the APPLBLK
} PARMBLK;
```

If pb_prevstate and pb_currstate are the same then the AES is drawing the object, not changing it.

Hints & Tips on Resources

Using ProgDefs

If a loaded resource file contains any ProgDef objects, their ob_spec; field will not be initialised on loading - this is up to the programmer. An APPLBLK structure needs to be allocated and initialised, then a pointer to it planted in the relevant ob_spec field.

The drawing routine (in the ab_code field) will then be called whenever that object needs drawing or changing (remember that if you have a ProgDef in a menu this may occur at *any* time). The routine called should normally be declared as both __stdargs and __saveds, taking a single parameter pointing to a PARMBLK. Hence a typical declaration would be:

```
int __stdargs __saveds my_progdef(PARMBLK *pb);
```

When your custom drawing routine has finished, the value it returns is the ob_state value which you wish the AES to render over your object, i.e. returning a value of 0 applies no extra effects, whereas returning CROSSED (for instance) would draw a cross over the object. Note that any number of ob_state values may be ORed together to produce the desired effect..

When designing ProgDefs it is often easiest to base them on existing objects which can be manipulated in WERCS, e.g. in the example program we implement a rounded button based on the normal square button, hence the text may be manipulated from within WERCS.

When the AES calls your drawing code you are still in the AES's 'context', i.e. you are using its stack, hence recursive routines or large local arrays may cause it to overflow. Note that this also means that the routine which is called *must* be compiled with stack checks *off* (the -v option). Also note that the AES is *not* re-entrant hence you may not make any calls to it (although you can, and should, call the VDI).

If you draw using the AES's handle (as in our example) then you should ensure that you maintain any of the VDI attributes, alternatively you may use your own virtual workstation which will avoid these problems.

Creating New Desktops

It is possible to replace the standard GEM background pattern (the area of the screen not used by the menu bar) using a special tree. This allows different colours and fill patterns to be used, as well as allowing icons to appear on the desktop.

A Form should be created in WERCS with the root object being a borderless Box with a suitable fill pattern and colour. If any icons are required these should be added to this Form. The size of the Form is not relevant. To tell GEM to use this Form, the size and position (ob_x, ob_y, ob_width and ob_height) fields in the root object should be set to the usable screen size, found using the AES wind_get(DESK, WF_WORKXYWH, ... call. The form can then be installed using a wind_set(WF_NEWDESK, ... call with an object parameter of zero. Before your program terminates, the desktop must be de-installed by passing a NULL value to the same call.

Note that installing a desktop does not cause it to be drawn and you should normally call form_do(FMD_FINISH, ... to force a redraw of the area.

Common Mistakes and how to avoid them

The following is a list of common mistakes made when programming with GEM and resources in general. The reasons given here are brief as there is insufficient space to expand upon them; they act as pointers for where to look in other documentation.

Problem: My program was mainly working but now it crashes during its initialisation.

Reason: Your resource file is out of step with your program and what was the Menu that you were displaying is now a Form; as a result the GEM menu_bar call bombs. Re-compile all the parts of your program that rely on the header file.

Problem: My dialog box doesn't disappear after you click on OK.

Reason: The dialog box is on top of one of your windows and you are not replying to WM_REDRAW events. If you don't open a window, the Desktop will re-draw the desktop tree for you automatically .

Problem: My program crashes when it should be displaying a Dialog Box.

Reason 1: If you have no editable fields and are passing -1 as the starting object the machine may crash, despite what some documentation says. Use 0 instead.

Reason 2: If you do have editable text fields make sure that they conform to the rules under FBoxText regarding editable text.

Problem: A GEM program crashes unexpectedly. After rebooting, the same program works correctly under the same conditions.

Reason: A program has modified GEM's data structures unintentionally. There are many possible ways of doing this; one to look out for is not doing a v_clsvwk after a v_opnvwk; that is leaving a Virtual Workstation open.

Problem: The mouse disappears or leaves extra pixels on the screen ('mouse droppings').

Reason: Your graf_mouse calls are mis-balanced in some way. For each hide (M_ON) call you *must* have a show (M_OFF) call.

Problem: There are mouse droppings where a menu has been pulled down.

Reason: You are not using wind_update (BEG_UPDATE, ... and graf_mouse (or the VDI v_hide_c) before writing to the screen.

Problem: When using some desk accessories the screen display is messed up.

Reason: Make sure that you are taking note of WM_REDRAW events and only updating the areas given by the wind_get (WF_FIRSTXYWH, ... and wind_get (WF_NEXTXYWH, ... calls. To test this, move a desk accessory about the screen; the Control Panel and the Saved! desk accessory can both be used.

Problem: Some desk accessories 'lose' their mouse when invoked from my program.

Reason: Make sure you don't remove the mouse until *after* you have done a wind_update (BEG_UPDATE, ... call and make sure that it is visible before calling wind_update for END_UPDATE.

Problem: The program works fine in medium and high resolution, but crashes when accessing a menu on 'old' ROM machine.

Reason: Your menu is taking up more than one quarter of the screen. When running in Low Resolution, a menu may not contain more than 16000 pixels. If you are using large menus, you may wish to consider using a special menu for low resolution, as WERCS does.

WERCS Language Details

This section details the language specific details of the name files produced by WERCS.

Assembly Language

If you have selected Assembler from the Language dialog box WERCS will produce a file with extension .I containing EQU statements of the form:

```
label EQU 1
```

If you have a saved a resource file called TEST.RSC you would then include the constants from the name file using:

```
INCLUDE TEST.I
```

The characters allowed in names are:

A-Z, a-z and _ as the first character and:

A-Z, a-z, 0-9, _ and . in subsequent characters.

Although designed with DevpacST in mind, the .I file can be used with other assemblers that follow the Motorola standard.

BASIC

If you have selected BASIC from the Language dialog box WERCS will produce a file with extension .BH containing CONST statements of the form:

```
CONST label%=1
```

If you have a saved a resource file called TEST.RSC you would then include the constants from the name file using:

```
rem $include test.bh
```

The characters allowed in names are:

A-Z, a-z as the first character and:

A-Z, a-z, 0-9, _ and . in subsequent characters.

The .BH file is designed with HISoft BASIC and Power BASIC in mind, and adaptation to other BASICs for the Atari ST is straightforward straightforward so long as the BASIC is suitable for serious GEM work.

C

If you have selected C from the Language dialog box WERCS will produce a file with extension .H containing #define pre-processor statements of the form:

```
#define label 1
```

If you have a saved a resource file called TEST.RSC you would then include the constants from the name file using:

```
#include "TEST.H"
```

The characters allowed in names are:

A-Z, a-z and _ as the first character and:

A-Z, a-z, 0-9, and _ in subsequent characters.

All sixteen characters of the name are significant as is the case with Lattice C 5. This may cause problems with some other C implementations, such as the HISoft C Interpreter, if your names have the first eight characters the same.

FORTRAN

If you have selected FORTRAN from the Language dialog box WERCS will produce a file with extension .INC containing PARAMETER definitions of the form:

```
INTEGER*4 LABEL
PARAMETER (LABEL=1)
```

The characters allowed in names are:

A-Z, a-z as the first character and:

A-Z, a-z ,0-9, and _ in subsequent characters.

All sixteen characters of the name are significant as in Prospero FORTRAN; the WERCS output was designed for use with this compiler and with Prospero's GEM bindings; they may not be appropriate for other FORTRANs.

We would like to apologise for the lack of an example FORTRAN program; we do not have the necessary FORTRAN expertise to produce this.

Modula-2

If you have selected Modula from the Language dialog box WERCS will produce a file with extension .DEF containing a definition module. For example, if you have saved a resource file called TEST.RSC, the file will be of the form:

```
DEFINTION MODULE TEST;
CONST label=1;
.....
END TEST.
```

If you are writing a one-module program you can use an implementation module of the same name.

Otherwise, you will need to write a implementation module like this:

```
IMPLEMENTATION MODULE TEST;
END TEST.
```

To access the constants from your other modules use:

```
FROM TEST IMPORT label;
```

or

```
IMPORT TEST;
```

and then access the labels as, for example, TEST.label.

Remember not to use the same name as your main module if you are using this method.

The characters allowed in names are:

A-Z, a-z, $ and _ as the first character and:

A-Z, a-z, 0-9, $ and _ in subsequent characters.

All sixteen characters of the name are significant.

Although designed for use with FTL Modula-2, the name files may be used with any Modula-2 compiler that follows the Third Edition of Wirth's book.

Pascal

If you have selected Pascal from the Language dialog box WERCS will produce a file with extension .INC containing constant definitions of the form:

```
CONST
  label=1;
....
```

If you have a saved a resource file called TEST.RSC you would then include the constants from the name file using:

```
{$I TEST.INC }
```

The characters allowed in names are:

A-Z, a-z as the first character and:

A-Z, a-z, 0-9, and _ in subsequent characters.

All sixteen characters of the name are significant.

Although our example programs are for Personal Pascal, the .INC files generated are suitable for use with most Pascal compilers.

The WTEST Example Programs

Compiling WTEST

To illustrate the most common resource-handling programming requirements, we supply an example program written in a variety of languages. The programs all do the same thing but the implementations vary according to the language used. A ready-to-run version called WTEST.PRG is supplied on the master disks; it needs WRSC.RSC to run.

To re-compile WTEST first you need to run WERCS and then use the Load command from the File menu to load WRSC.RSC from your backup disk. When WRSC.RSC is loaded the name file, WRSC.HRD containing the names of the forms and objects will be loaded as well.

All the following commands are on the File menu. Click on Language and then select the language that you wish to use by clicking on the appropriate radio button in the dialog box. Next save the file using Save; this will re-save the WRSC.RSC resource file, the name file WRSC.HRD and also a file for the language of your choice. Next use Quit to leave WERCS.

One more general point: if you edit a resource file and change its structure you will need to re-compile your program, in case any constants have changed.

C

WTEST.C is the source file for use with Lattice C 5. This is an extended version of the standard WTEST, only supplied with Lattice C 5, which rather than using square buttons, uses rounded ones. Several other headers files supplied as part of Lattice C 5 are also used.

Assembly Language

WTEST.S is the source file for use with DevpacST2. As well as the WRSC.I file produced by WERCS you will need GEMMACRO.S and AESLIB.S from your DevpacST master disk.

BASIC

WTEST.BAS is the HISoft BASIC version which will also compile under Power BASIC. It needs the GEMAES.BH file from your BASIC master disk as well as the WRSC.BH file produced by WERCS.

Modula-2

WTEST.MOD is the FTL Modula-2 version. You will need to compile the file WRSC.DEF produced by WERCS before you compile WTEST.MOD. You will also need to compile WRSC.MOD from your WERCS backup disk before you link.

Pascal

WTEST.PAS is the Personal Pascal version; to compile this you will need GEMSUBS.PAS from your Personal Pascal backup disk.

WTEST structure

The different versions of WTEST all do the same thing but the implementations vary according to the language used. We have tried to keep variable names and procedure/function names as consistent as possible.

The program is deliberately over-simplified; it manages to avoid calling the VDI completely and gets away with an evnt_mesag, avoiding the dreaded evnt_multi. The general structure of the code in all the programs is as follows:

Procedure INITIALISE

This does the required GEM initialisation then loads the resource file. The tree address of the menu is found and the menu installed. The usable screen size is found and certain global variables initialised.

Procedure SETDESK

This sets the new desktop pattern to be a particular address and forces the AES to re-draw the whole screen.

Procedure DEINITIALISE

Resets any installed desktop, removes the menu bar, frees the resource, then does any required GEM de-initialisation.

Procedure HANDLE_DIALOG

A general dialog box handler which starts by centring and drawing the box. User interaction is handled by form_do and, on return, if the exit object was a Button, it is de-selected.

Procedure SET_TEDINFO

This allows a particular TEDINFO structure to have its data portion set to a particular string.

Procedure GET_TEDINFO

Allows a particular TEDINFO structure to return its data portion.

Procedure ROUND_BUTTON

This is the code that the ProgDef in the Lattice C 5 version of WTEST uses.

Procedure OBJ_INIT

This modifies the button objects in the dialog box to be ProgDefs. The strings already there are saved and used subsequently for the round buttons. It also determines the AES's VDI handle for the use of ROUND_BUTTON. This routine is only supplied in the Lattice C 5 version.

Procedure SET_BUTTON

This allows one particular radio Button to be set from a group. If invalid parameters are specified the routine will never finish.

Procedure GET_BUTTON

Allows a group of radio Buttons to be interrogated to see which is selected.

Procedure TEST_DIALOG

This handles the particular dialog box in the resource file. It implements proper cancelling - that is, if it is cancelled, the Button state and Text entry are left alone.

Procedure HANDLE_MENU

This is the menu-click dispatcher; it takes various actions, depending on which menu item has been clicked, and also de-selects the menu title.

Procedure MAIN

This is the main loop, acting only on MN_SELECTED message events. In a proper program evnt_multi would be used and a far greater selection of cases would have to be dealt with.

HRD file format

.HRD files consist of a header record, any number of variable length data records and then an end-of-file record.

HRD Header record

Name	Size	Meaning
version	word	1 at present
autonaming	byte	1 if auto-naming selected 0 if no auto-naming
langflag	byte	1 if C, 2 if Pascal, 4 if Modula-2, 8 if FORTRAN, 16 if assembler, 32 if BASIC
autosnap	byte	0 if no character-snap 1 if half character-snap 2 if full character-snap
casing	byte	0 if mixed 1 if upper 2 if lower
autosizing	byte	1 if auto-sizing 0 if no auto-sizing
reserved	byte	not used at present

HRD Data Record

Name	Size	Meaning
type	byte	0 if Form, 1 if Menu, 2 if Alert, 3 if Free String, 4 if Free Image, 5 if object (rather than tree), 6 if end-of-file record, 7 if record names a prefix rather than a name.
reserved	byte	not used at present
treeIndex	word	number of tree
objIndex	word	if object then object number within tree
name	varies	Name terminated with a single null

LNG file format

The WERCS.LNG file is a text file containing the information that WERCS uses to work out which name file to produce, what to output and what is a valid name. You can modify this if you wish; though be warned that it is easy to produce files that your compiler won't like.

The file consists of a number of Language specifications followed by an end record. Each Language specification starts with a line like:

```
*LANGUAGE C
```

where C is the name of the language. This is used purely for documentation purposes; it won't affect the Language Dialog Box for example. The records are for C, Pascal, Modula-2, FORTRAN, assembler, and BASIC in that order.

The end record is a single line:

```
*END
```

and should be the last line in the file; the information following it will be ignored.

The other lines in the language specification may appear in any order and can be one of:

```
*SOURCE .H
```

This specifies the extension of the source file that WERCS will generate (.H in this example); the full stop (.) must be present.

```
*SIGNIFICANCE 16
```

Specifies the number of significant characters in names; minimum 1 maximum 16. The default is 16.

```
*INITIAL a-z,A-Z,_
```

Specifies the characters that are allowed as the first letter of the name. The hyphens (-) indicate a range of letters. The commas may be omitted. The equivalent of this would be

```
*INITIAL a-zA-Z_
```

which is far less obvious. Do not put spaces in the INITIAL string. The default is a-z, A-Z.

```
*FOLLOW a-Z,A-Z,0-9,_
```

specifies which characters are allowed in names other than in the first position. The syntax to specify the characters is the same as for *INITIAL. Default is a-z, A-Z, 0-9.

```
*INIT
*TREE
*OBJECT
*EXIT
```

These commands specify the text that is generated in the source file. The *INIT text is generated once at the top of the file, *END at the end of the file, *TREE for each named tree in the file and *OBJECT for each named object. Each entry may be more than one line long, the entry being terminated by the next * command. The following special pairs of characters may be used:

Character	Meaning
% F	base file name (without drive, directory or extension),
% T	name of the current tree,
% N	name of the current object,
% V	the value of the current object (for *OBJECT) or tree (for *TREE),
% %	a single % character.

Of the above only %F and %% should be used in *INIT and *EXIT. The default for *INIT, *EXIT, *OBJECT and *TREE is no text at all.

See the WERCS.LNG file for an example. If you want to generate code for a language that is not supported already, modify the definition of another language that you do not use. If you are adventurous you can even change the name of the Button in the Language box in WERCS.RSC but if something goes wrong don't blame us!

Appendix C Converting to Lattice C 5

Lattice 3.04

Old Lattice 3.04 programs are probably the easiest programs to convert to Lattice C 5. Often few or no changes will be required. There have however been many changes to gain ANSI conformance on the runtime libraries and some rationalisation of names to match those used by Lattice C on other architectures. The important changes for upgrade users are:

- abs has been changed so that it only deals with the type int.
- malloc, calloc, free, realloc have been extended to respond correctly to passing of ANSI NULL arguments. Note that this means that *any* use of malloc(0) will return a NULL pointer. Also note that free returns the 'type' void, not int as with the previous release.
- An extra level of data hiding has been introduced between the ANSI file and GEMDOS file handling. In particular fileno *no longer* returns a GEMDOS file handle, but instead an *internal* handle. The _chkufb function may be used to obtain the mapping from internal to GEMDOS handles.
- dcreat, dclose, dopen, dread, dwrite and dseek have had an underscore prefix added. Note also that the level of functionally at this level is significantly greater than before, as many of the GEMDOS anomalies are removed at this level.
- ^Z is no longer recognised as end of file marker in text files; this usage was not widespread and invariably caused confusion.
- lsdata, lsdptr, lsstatic, lsauto, lsheap and lspptr were hang-overs from the MS-DOS implementation in version 3 and not strictly relevant to the 68000 environment and so have been removed.
- Almost all internal variables have been renamed or removed, any references to such variables should (in general) be removed.
- The values placed in _OSERR are now the positive GEMDOS error codes. This change was to increase conformity with other Lattice compilers.

- allmem and bldmem are no longer relevant to the new dynamic memory manager (cf. the old static memory manager); any calls to these functions are best removed.

- The exec family of functions are no longer available, they did not perform as described under the old documentation anyway (in fact they were fork synonyms). TOS is incapable of performing exactly the operations required to implement these functions as previously described.

- The -n flag on the compiler has the opposite sense to that under 3.04, i.e. -n *truncates* to eight characters rather than increasing the significance to 32 characters.

- The Line-A functions have *all* been implemented, and a correct header file made available together with many extensions. In particular, the showmouse macro from linea.h has been corrected to include the hide depth parameter; refer to the Line-A documentation for details.

- In conformance with ANSI, memcpy is not guaranteed to perform correctly when blocks overlap. Under version 3 this case was handled correctly.

- _MNEED is defunct due to the dynamic memory model used. It may be used to specify an initial heap size (e.g. if you have a program which will have only very small malloc requirements) although it no longer specifies the maximum heap size. In particular negative values are ignored.

- The GEM AES and GEM VDI header files have been split into two, although both may be included via #include <gemlib.h>.

- Many of the types in the AES and VDI prototypes have been changed to reflect more natural type usage. This should not, however, affect the code generated.

- The AES, VDI and Line-A libraries were re-written from scratch, hence any bugs in the old libraries are most unlikely to exist in the new libraries.

Other Differences

This section lists other differences from version 3; hopefully all important conversion differences are listed above. Here is the list of the improvements and changes in Version 5, relative to Version 3:

- The compiler now uses sequence points to ensure correct evaluation and side effect generation according to the standard.

- The compiler now includes a full ANSI pre-processor with string facilities, token facilities and appropriate scoping of substitution symbols. The defined() directive is also supported. In addition, __DATE__ and __TIME__ provide the date and time of compilation.

- The const and volatile keywords are supported.

- Function prototypes may now include an optional parameter name. Also functions taking a variable number of arguments may be indicated with ellipses '...'.

- String literals may now be concatenated to allow easier coding of long strings.

- The cast operation (void *) correctly coerces a type without any warning.

- Many diagnostic messages have been added to detect programs that do not conform to the ANSI standard.

- The compiler now recognises several new keywords:

signed	Overrides any default unsigned options.
near	Declares a data item to be addressed relative to the global base register. When used with a subroutine, it indicates a PC-relative subroutine call.
far	Declares an item that must be addressed with a full 32 bit address.
huge	Same as far.
__regargs	Defines a subroutine that is to be called with register parameters.
__stdargs	Defines a subroutine that is to be called with standard stack parameters.
__asm	Defines a subroutine that takes its parameters in a specific register
__saveds	Defines a subroutine that is to load up the global base pointer upon entry.
__interrupt	Defines a subroutine that may be called from interrupt code.

- To provide faster compilation of a large project, the symbol table may be saved out to disk so that it can be used in future compilations. If you run a large header file through the compiler and save it in this way, subsequent compilations using that header file will be much faster.

- The compiler provides an option to ignore redundant #include statements (i.e. several #include statements that refer to the same file).

- The search rules for #include files have been modified to conform to standard UNIX search conventions.

- It is now possible to disable particular error messages as well as to change the severity of most messages. Along with this, it is now possible to specify a maximum number of errors allowed in a compilation so that the compiler will abort.

- We have implemented an improved form of error recovery for many of the common mistakes to eliminate many of the situations that resulted in the cascade of errors.

- All of the compiler messages have been moved to a separate file to simplify adaptation of the compiler to non English language environments.

- In order to produce a compiler that fits well on a smaller system, we have elected to provide a big version of the compiler that includes some additional features. If you wish to take advantage of these features (at a cost of about 15K), you must use this big compiler instead of the standard one.

- The big compiler provides a full listing ability including macro expansion display, nest level counting, and include file listing. This listing may also include an optional cross reference of all variables, #define values and structure tags.

- The big version of the compiler may be used to generate prototype files of all functions encountered in a module. This eliminates the potentially tedious task of constructing the list of prototypes for all functions in a project.

- The compiler generates instructions optimised for each of the 680x0 family processors including support for the address modes found on the 68020 and 68030.

- The compiler provides an option to generate in-line floating point instructions which directly access the 68881 and 68882 maths co-processors. This code takes advantage of register tracking.

- You may now instruct the compiler to choose code sequences optimised for space or for time.

- In addition to the -r0/-r1 flags, you may freely mix the style of subroutine calls with the near and far keywords. Those declared near will be referenced with the pc-relative addressing while far will use the full 32-bit addressing.

- Data may be addressed much more freely with the near and far keywords. These control the type of addressing to be used for external data. Only those items declared near will be addressed as a 16-bit offset from the A4 register. All others will be accessed with a full 32-bit address.

- When the -cs option is used, string constants will be placed in the code section. This option is beneficial when using the -b0 option.

- Two styles of register parameters are supported. The -rr option causes the compiler to place, automatically, up to four parameters in registers for subroutine calls. The __asm keyword may be used in conjunction with a register specification list to cause the compiler to pass parameters in a given register.

- The compiler now tracks the condition codes affected by the generated code in an attempt to avoid generating unnecessary test instructions.

- Stack cleanup on subroutine calls is delayed as long as possible to allow coalescing and even elimination of the cleanup across multiple calls.

- The bitwise Boolean operations generate better code for dealing with constant values.

- Division by 2, 4, or 8 no longer generates a subroutine call. The compiler generates inline code to normalise and perform the calculation.

- Bit shift operations have been re-written completely. The compiler now generates optimal shift sequences for all constant values.

- The compiler attempts to place as many variables as possible in registers unless this feature is explicitly disabled.

- The code generator now takes full advantage of *all* 68000 address modes including auto increment and PC-relative indexed. Tracking of indexing operations allows the compiler to suppress unnecessary additions and substitute indexed address modes.

- Loading of specific constants has been optimised to generate the optimal code sequence and avoid MOVE.L # as much as possible.

- Several new built-in functions have been added:

abs	Return the absolute value of an integer.
__emit	Insert a hex word into the instruction stream.
..._fpc	Generate MC68881 transcendental operation.
geta4	Force loading of the global data register.
getreg	Obtain the contents of a specific 68000 register.
max	Return the larger of two integers.
memcmp	Compare memory blocks.
memcpy	Copy memory block.
memset	Initialise memory block.
min	Return the smaller of two integers.
putreg	Directly store into a specific 68000 register.
strcmp	Compare strings.
strcpy	Copy strings.
strlen	Obtain string length.

- Many small code optimisations suggested by users have been implemented. In particular, the compiler no longer will generate NOP instructions. Also, MOVEM instructions in the prologue/epilogue that reference a single register are converted to MOVE instructions.

- All GEMDOS/BIOS/XBIOS functions are available via inline TRAPs eliminating the overhead of 'stub' based methods.

- switch statements on values which are in the range of a short are converted to use the more efficient code.

- The entire run-time library has been re-written to take advantage of better algorithms. The most important routines were recoded in assembly language.

HiSoft C

Converting HiSoft C programs to Lattice C 5 is normally fairly simple, requiring few changes to the source code. The following options are recommended to ensure compatibility with the HiSoft C model:

-au Force all characters to unsigned; HiSoft C always considers plain characters to be unsigned. The use of this option will ensure that any dependencies on this behaviour will not cause unexpected effects.

-fd Force all floating point declarations to be of type double as HiSoft C only supports the double real format.

The following points should also be taken into account when converting code:

- If you are using the HiSoft C toolbox the main function should be renamed as lc_main, so that the special initialisation code runs before your program.
- The HiSoft C toolbox is supplied already compiled as a library for use with Lattice C 5 and can simply be linked in - see the **Installation Guide**.
- HiSoft C programs which do not use the GEM toolbox, but do use GEM, *must* use appl_init and appl_exit. Their use is optional under HiSoft C, whereas Lattice C 5 uses these calls to perform some initialisation itself. Failure to call these will result in mysterious crashes.
- The HiSoft C functions strgetfn and strsplfn were re-named from their traditional Lattice names. The functions strmfn and strsfn respectively should be used in their place. The easiest solution is to define this mapping on the command line to Lattice C (since HiSoft C would not allow the mapping within the program) viz:

```
-dstrgetfn=strmfn -dstrsplfn=strsfn
```

Where code cannot be made to run successfully under both systems, the IC and LATTICE pre-processor symbols can be used to distinguish between the translation environment; HiSoft C always defines IC, Lattice C 5 always defines LATTICE. Hence one can write:

```
#ifdef IC
  /* HiSoft C specific code */
#else
  /* Lattice C version of the code */
#endif
```

Appendix D
GST Support

LinkST, The GST format linker

Introduction

LInkST is a linker that links GST-format files. In general we recommend that you use the Lattice C Linker,CLink rather than LInkST since CLInk is faster and has more facilities. LInkST does have one advantage over CLInk though: it will let you link with other languages that produce GST format files.

For example, if you have some assembly language written with GenST, the assembler supplied as part of HiSoft DevpacST2, then you could use GenST to produce a GST-format file and then link it, together with your C code and the GST libraries, to produce an executable program. If the assembly language module you have is short or doesn't use the extensions to the Motorola standard provided by DevpacST, we recommend that you convert your assembly language to the Lattice assembler, asm, so that you *can* use CLInk.

If you wish to link your Lattice C code with code written in another high level language that supports the GST format (producing a so-called *mixed language* program) or you wish to use a third party library provided in GST format, then you will need to use LInkST.

Producing mixed language programs is not for the faint hearted; you need to ensure that the required start up code is provided for both languages and that the memory models, including register conventions,are compatible. Fortunately Lattice C 5 provides a wealth of options, so that you can match most popular conventions. See the section on **LC, The Compiler**.

If you are 'only' linking with assembly language, things are much simpler since you can modify the code that is being called to handle whichever memory model you wish to use.

Note that Lattice C 5 does not produce the other commonly used format on the ST, DRI linkable code, because this linker format is not sufficiently rich to support some of code constructs that Lattice C 5 generates.

LInkST will only link GST-format files.

Compiling code in GST format

To produce code that can be linked with LInkST you will need to use the LC.TTP driver's -z option or, alternatively, convert a Lattice format file to GST format by using lc2gst as described at the end of this appendix.

If you are producing a program that is larger than about 32K using GST format then you cannot use the compiler's default small code model because GST linkers cannot generate ALVs, the clever method that CLInk uses to allow you to use the small code model with large programs. Thus you will need to use the -r0 option when compiling GST-format libraries.

The GST libraries that we supply follow the same naming conventions as those in Lattice format only with the GST .bln extension. If you are using -r0, you'll need to use lcnb.bln, lcgnb.bln and lcmnb.bln as your libraries, which give you large code and large data. Thus to link with these you will also need to use the -b0 compiler flag.

There are a few other points regarding the use of GST format:

- The maximum amount of near data is restricted to 32K rather than 64K as with CLInk.
- Lattice line number debugging information cannot be included in your executable program.
- There are some errors that CLInk can spot that LInkST cannot; these are mainly mis-uses of memory models.

Invoking LinkST

As with most of the tools supplied with Lattice C 5 you can either run LInkST from the Desktop or from a shell and can either supply a complete command line or specify a link file which contains the required information.

The command line should be of the form:

```
<filename> <-options> [filename] [-options]
```

Options are denoted by a - sign then an alphabetic character, supported options being:

-B generate a true BSS section for any such named sections

-D debug - include all symbols in the binary file using DR standard 8 character format (for MonST2C or other debuggers)

-F force pass 2 of the linker, useful if you want to see all errors (as any pass 1 errors will, by default, stop the link before the second pass)

-L specify that all following filenames are library filenames

-M dump a map file showing the order of the sections and labels. The map filename will be the main filename with an extension of .MAP

-O specify object code filename, may be followed by white space before filename

-Q 'quiet' mode, which disables the pause after the link

-S dump a symbol table listing, The symbol table filename will be the main filename with an extension of .SYM

-T truncate to eight characters

-U force upper case

-W specify control file filename, defaults to .LNK extension

-X extended debug, using the HiSoft Extended Debug format for use with MonST2C.

Normally any file specified given are assumed to be input files, defaulting to the extension of .BIN, though if a .LNK extension is specified it will be taken to be a control file. After a -L option, filenames are all assumed to be library files.

The output file can be specified with the -O option on the command line, or using the OUTPUT directive in the control file. If there is more than one of these directives or options, the last one is used. If there is none given, then the first input filename specified in the command line or control file is used, with an extension of .PRG.

Example Command Lines

```
PART1 PART2 -d
```

Reads PART1.BIN and PART2.BIN as input files, and generates PART1.PRG as an output file complete with debugging information.

```
PART1 PART2 -o TEST.PRG
```

Reads PART1.BIN and PART2.BIN as input files, and generates TEST.PRG as an output file.

```
-o TEST.TOS START -l MYLIB -s
```

Reads START.BIN as an input file, selectively reads MYLIB.BIN as a library, and generates the output file TEST.TOS and the symbol listing file TEST.SYM.

LinkST Running

LInkST has two passes - during pass 1 it builds up a symbol table of all sections and modules, and during pass 2 it actually creates the output file. When it starts it prints a logon message, then reports on which files it is reading or scanning during both passes. This gives you some idea of what takes time to do, as well as exactly where errors have occurred.

If there is enough free memory at the end of pass 1, LInkST will use a cache to store the output file, which speeds up the process greatly. If it uses the cache it will write to the disk at the end of pass 2, and report the number of errors.

When the link finishes you will be prompted to press a key before quitting. This is to give you an opportunity to read any warning or error messages before returning to the Desktop. You can disable this pause by using the -q option, useful if you are using a CLI .

Error and warning messages are directed to the screen - if you want to pause output you can press Ctrl-S, while Ctrl-Q will resume. Pressing Ctrl-C will abort the linker immediately.

You can re-direct screen output to a disk file by starting the command line with

```
>FILENAME.TXT
```

or you can re-direct it to a printer by starting the command line with

```
>PRN:
```

to the parallel port, or

```
>AUX:
```

to the serial port.

If you do re-direct output in this way you should use the -q option as you won't be able to see the prompt at the end of the linking.

Control Files

The alternative way to run the linker is to have a control file for the programs which you are linking together.

If you require a lot of options which won't fit on the command line or you get bored of typing them, you can use a control file, which is a text file containing commands and filenames for the linker. The default extension is .LNK and the control filename is specified on the command line using the -w (for With) option. Each line can be one of the following:

```
INPUT <filename>
```

This specifies a filename to be read as an input file. The default extension is .BIN if none is given.

```
OUTPUT <filename>
```

This specifies the filename to be used for the output file. There is no default extension - you must specify it explicitly.

```
LIBRARY <filename>
```

This specifies a filename to be scanned as a library. The default extension is .BIN if none is given.

```
SECTION <sectionname>
```

This allows specific section ordering to be forced.

```
DEBUG
```

All symbol names included in the link are put in the output file so that debugging programs such as MonST2C can use them when the program is running.

```
XDEBUG
```

Similar to the debug option but uses HiSoft Extended Debug format for up to 22 character significance.

```
DATA size[K]
```

The BSS segment size is set accordingly. The size can be given either as a number of bytes or as a number of K-bytes (units of 1024). This option is particularly useful for the Prospero compilers which effectively use the BSS segment for their stack.

```
BSS <sectionname>
```

Specifies that the named section should lie in the GEMDOS BSS section area. This can save valuable disk space, but will generate errors if the section contains any non-zero data. This should not be used at the same time as the DATA statement.

```
TRUNCATE
```

Causes all symbols to be truncated to 8 characters. This is sometimes required to link assembly language with long labels to high-level language code with short labels.

```
UPPER
```

Forces all symbols to be automatically upper-cased. This is sometimes required when a compiler or assembler generates case-insensitive code.

Blank lines in the control file are ignored, and comments can be included by making the first character in the line a *, a ; or a !.

With the INPUT or OUTPUT directive, if the filename is specified as * it is substituted the first filename on the command line. This can be useful for having a generic control file for linking C programs for a particular memory model.

An example control file is:

```
* control file for linking large model gem programs
INPUT CNB
INPUT *
XDEBUG
LIBRARY LCGNB
LIBRARY LCNB
SECTION TEXT
SECTION DATA
BSS UDATA
```

Assuming this control file is called CPROG.LNK, the LInkST command line

```
TEST -w CPROG
```

will read as input files CNB.BIN and TEST.BIN, and scan the libraries LCGNB.BIN and LCNB.BIN. The object code, including extended debug information, will be written to TEST.PRG, as no output file was explicitly specified.

The two SECTION directives, above, ensure that the TEXT and DATA sections appear in the correct order in the output file. The BSS directive ensures that the UDATA section is treated as a true BSS section.

If you do not specify a drive name in the control file or on the command line, the default drive will be assumed. If you run LInkST from the Desktop, the default drive will always be the same as that containing the file on which you double-clicked; though if you run it from a CLI or from the editor this will not necessarily be so.

LinkST Warnings

Warnings are messages indicating that something might be wrong, but probably nothing too serious.

duplicate definition of value for symbol <x>

The symbol was defined twice. This can happen if you replace a subroutine in a module with one of your own, for example. The linker will use the first definition it comes across, and give this warning on the second.

module name is too long

Module names can only be 80 characters long.

comment is too long

Comment directives are only allowed to be 80 characters long (don't ask us why, we don't know!).

absolute sections overlap

Two absolute sections clash with each other.

SECTION <x> is neither COMMON nor SECTION

A section name was specified without defining its type.

LinkST General Errors

unresolved symbol <x> in file <x>

The symbol was referred to but not defined in the file. There may also be other files which refer to the symbol, but this error gives you a start in your search!

XREF value truncated

A value was too large to fit into the space allocated for it, for example a BSR to an external may be out of range.

bad control line <x>

An illegal line was found in a control file.

non-zero data in BSS section

A section wanted as a true BSS section contained non-zero data.

LinkST Input/Output (I/O) Errors

file <x> not found

Can't open output file <x>

Can't open map file <x>

Can't open symbol file <x>

Can't open input file <x>

i/o error on input file

disk write failed

filename <x> was too long

LinkST Binary File Errors

These are errors in the internal syntax of the input file, and should not occur. If they do it probably means the compiler, assembler or converter produced incorrect code.

missing SOURCE directive

Can occur if a file is not in GST format, for example a DRI file.

runtime relocation is only available for LONGs

attempt to redefine id of symbol <x>.

attempt to DEFINE <x> with <id> of zero

bad operator code 0x99 in XREF directive

bad truncation rule in XREF

wrongly placed SOURCE directive

bad directive <99>

<id> <99>not DEFINEd as a SECTION but used as one

attempted re-use of <id> <99>as SECTION id

attempted re-use of <x> as SECTION name

section is COMMON but being used as though it's not

SECTION is being misused as COMMON

unexpected end of input file

'Linker Bug' Messages

These can be produced as a result of internal checks by the linker. If you get one please send us copies of the files you are trying to link!

GSTlib, The GST format librarian

GSTlib is a librarian that is designed for maintaining GST format libraries. When specifying filenames to GSTlib you must explicitly include the file extension, normally .bln.

GSTlib has a number of different possible command lines as shown below:

Replace modules

```
gstlib r[vsq][a|b obmod] library [files...]
```

This will replace any current occurrences of the modules contained in the given files. If a module is not already present in the library then it will be added. If the library does not exist then a new library will be created that just contains these files. The following additional options may be used

v	(verbose). Echo when adding or replacing a module or creating a library.
a mod	(after). If adding a new module insert it after mod.
b mod	(before).If adding a new module insert it before mod.
s	(sort). Sort the library so that it can be scanned by a linker in a single pass, if possible. If this fails due to circular references then the new library will be saved in libname.tmp and the original library left as it was.
q	(quit). Wait for Return to be pressed before exiting GSTlib, for use with the Desktop.

Update modules

```
gstlib u[vsq][a|b obmod] library [files...]
```

This works in precisely the same way as the replace modules option except that the modules are only updated if the versions in the files are more up to date than the version in the library. Exactly the same options may be used as with the r option.

Load modules

```
gstlib l[vq] prog.lib [files...]
```

This option can be used to produce a library that contains just the modules that would be included when linking a program. The v and q modifiers have their usual meaning. For example:

```
gstlib l prog.bin c.bin yourprog.bin lc.bin
```

could be used to create a library containing the modules required by yourprog. You could then run LInkST without the need to scan the library. Once you have created a library using this option it is unwise to sort it subsequently.

Delete modules

```
gstlib d[vsq] library [modules....]
```

Deletes the modules with the given names. As usual v will cause the librarian to inform you of its progress, q will pause before exiting and s will attempt to sort the library after performing the deletions.

Move modules

```
gstlib m[vsq][a|b obmod] library [modules...]
```

Moves the given modules to thr end of the library unless either a or b are specified, in which case:

a mod moves the modules immediately after obmod and

b omod moves the modules immediately before obmod.

The other modifiers have their usual meanings.

Tabulate modules

```
gstlib t[vvvvvq] library [modules...]
```

This form of command line displays information about the given modules in the library. If no module list is included, the information is given about all modules. The different levels of information are as follows:

v module names only

vv as per v and the size and date

vvv as per vv and the list of exported (xdef) symbols

vvvv as per vvv and the list of imported (xref) symbols

vvvvv as per vvvv and a cross reference of the symbols

As ever, q may be included to pause before GSTlib terminates.

Extract modules

```
gstlib x[vkq] library [modules....]
```

This extracts the given modules from the library. If no modules are specified then all the modules are extracted. Note that module names may have .o, .c or .bin appended to them depending on the tool that produced them. The additional modifiers are:

k keep date stamp from library on the extracted files

v inform the user of progress

q pause before exiting

If a module that is being extracted does not have a name it will be placed in a file called dummy###.BIN where ### is a unique decimal number.

Librarian command files

If the command line to GSTlib contains an @ sign, the name following this is taken as a command file name and the command read from this. Additionally such files may contain lines starting with #; these are treated as comments. Long lines may be split by using \ as the last character of the line.

Example command lines

```
gstlib tv lc.bin
```

List all the modules in the library lc.bin; there are quite a few!

```
gstlib xk lc.bin printf.o
```

Extract the module printf.o into a file called printf.o retaining the date stamp that it had in the library.

```
gstlib d lc.bin printf.bin
```

Remove the module printf.bin from lc.bin.

The source code to GSTlib is supplied; see the **Installation Guide** for details.

lc2gst, The Object File Convertor

lc2gst is a tool for converting Lattice C object files to the GST format. It has three different forms of command line as follows:

```
lc2gst file.o
lc2gst -o dir\ file.o
lc2gst -o john file.o
```

The first form simply converts file.o to file.bin. Note that the .o *is* required.

The second form places the converted file in the directory dir; note that the terminating \ must be present. This option can be used to keep your GST format files in one directory.

The final form causes file.o to be converted to john.bin. Note that the .bin extension is always used, regardless of any explicit extension after the -o.

lc2gst can convert more than one file at once: simply include the other files to convert at the end of the command line. It does not, however, expand wildcards itself, so that you will need a shell that supports this, such as Craft, to provide this facility.

Appendix E
Quick Options Reference

LC	LC1	LC2	
-b	-b		Base relative data
-b0	-b0		Non-base relative data
-b1	-b1		Base relative data
-B			Always use 'big' compiler
-c+	-c+		Suppress structure messages
-ca	-ca		ANSI compatibility
-cc	-cc		Allow nested comments
-cd	-cd		Allow $ in identifiers
-ce	-ce		Suppress error line printing
-cf	-cf		Require function prototypes
-cg	-cg		Process ANSI trigraphs (not implemented)
-ci	-ci		Suppress multiple includes of same file
-ck	-ck		Allow new keywords
-cl	-cl		Forces long alignment of all external data
-cm	-cm		Allow multiple character constants
-co	-co		Enable old style preprocessor
-cq	-cq		Strengthen aggregate equivalence type checking
-cr	-cr		Allow register keywords
-cs	-cs		Create only one copy of identical strings
-ct	-ct		Enable warnings for tags used without definition
-cu	-cu		Force all char declarations as unsigned char
-cw	-cw		Shut off warning for return without a return value
-cx	-cx		Treat all global declarations as externals
-C			Continue on error
-d	-d		Enable debugging
-d0	-d0		Disable debugging

-d1	-d1		Enable debugging - dump line table
-d2	-d2		Generate symbol information
-d3	-d3		Generate symbol information, dump at every line
-d4	-d4		Generate full symbol information
-d5	-d5		Generate full symbol information, dump at every line
-dx=y	-dx=y		Define preprocessor symbol
-e	-e		Recognise extended character set
-e0	-e0		Japanese character set
-e1	-e1		Chinese character set
-e2	-e2		Korean character set
-Ex=y			Define environment variable x with value y
-f	-f		Use standard Lattice libraries
-f8	-f8		Generate code for Motorola 68881
-fa	-fa		Auto-detecting I/O based 68881
-fi	-fi		I/O based 68881 maths
-fl	-fl		Use standard Lattice libraries
-fd	-fd		Treat all declarations as double precision
-fm	-fm		Use float as single precision and double as double precision
-fs	-fs		Treat all declarations as single precision
-gd	-gd		Cross reference defined symbols
-gc	-gc		Cross reference compiler provided files
-ge	-ge		List excluded lines
-gh	-gh		List header files
-gi	-gi		List included files
-gm	-gm		List macro expansions
-gn	-gn		Print narrow lines
-gs	-gs		List source
-gx	-gx		Produce cross reference listing
-Hxxx	-hxxx		Read in header file xxx
-ix	-ix		Specify include directory
-j<n>	-j<n>		Disable message n
-j<n>e	-j<n>e		Make message n an error instead of a warning

-j<n>i	-j<n>i		Disable message n
-j<n>w	-j<n>w		Enable message n as a warning
-L+			Specify additional linker objects
-La			XADDSYM option
-Lb			BATCH option
-Lf			MAP option
-Lg			GEM library
-Lh			Hunk map option
-Ll			Library map option
-Lm			Lattice maths library
-Ln			NODEBUG option
-Lq			QUIET option
-Ls			Symbol map option
-Lv			Verbose option
-Lx			XREF map option
-l	-l		Align objects on longword boundaries
-M			Only compile modified source files
-m		-m	Generate code for Motorola 68000
-m0		-m0	Generate code for Motorola 68000
-m1		-m1	Generate code for Motorola 68010
-m2		-m2	Generate code for Motorola 68020
-m3		-m3	Generate code for Motorola 68030
-ma		-ma	Generate code for all Motorola processors
-mc		-mc	Disable cleanup overhead reduction enhancement
-mr		-mr	Disable automatic registerisation
-ms		-ms	Generate code optimised for space
-mt		-mt	Generate code optimised for time
-n	-n		Retain only 8 characters for identifiers
-ox		-ox	Place object file in location x
-p	-p		Preprocess only
-pe	-pe		Generate prototypes only for externs
-ph	-ph		Generated precompiled header file

-pp	-pp		Generate prototypes with __PROTO for portability
-pr	-pr		Generate prototype file
-ps	-ps		Generate prototypes only for static functions
-qx	-ox		Place quad file in location x
-q<n>e	-q<n>e		Quit compilation after n errors/warnings
-q<n>w	-q<n>w		Quit compilation after n warnings
-q			Same as -qlelw
-q-	-q-		Never quit on any errors or warnings
-r	-r		Default subroutine calls to near (PC-relative)
-r0	-r0		Default subroutine calls to far (Absolute)
-r1	-r1		Default subroutine calls to near (PC-Relative)
-rr	-rr		Default subroutine calls/entries to register conventions
-rs	-rs		Default subroutine calls/entries to stack conventions
-rb	-rb		Generate code for both register and stack convention entries
-Rx			Place compiled objects into library x
-s		-s	Specify default segment names
-sb=x		-sb=x	Specify name for BSS segment
-sc=x		-sc=x	Specify name for code segment
-sd=x		-sd=x	Specify name for data segment
-ta			Force linking of a desk accessory startup stub
-td			Force linking of auto detecting startup stub
-tr			Force linking of resident startup stub
-t=x			Use file x as the startup code
-u	-u		Undefine all preprocessor symbols
-ux	-ux		Undefine preprocessor symbol x
-v		-v	Disable stack checking code
-w	-w		Default to short integers
-x	-x		Treat all global declarations as externals
-y		-y	Load up A4 with base address at start of functions
-z			Generate GST-linkable code

Appendix F
The Lattice C Start-Up

Introduction

The Lattice C compiler is supplied with a wide range of differing startup stubs which perform various levels of initialisation and have differing impacts on the programs which may be written.

The stubs

4 different stubs are provided for the following purposes:

Standard - suffix none

This family of stubs provide the normal entry and exit code required by a GEM or TOS program designed to be run either from the Desktop or from another program via GEMDOS Pexec. They perform full command line parsing via either the Atari extended command line method, or if this is not available via the command line embedded in the programs basepage. The memory available to the program is assessed and various internal variables are initialised to support the dynamic memory allocator (note that this allocator is careful to ensure that bugs in TOS pre-1.4 are not aggravated). The environment variables available are also parsed into the standard UNIX environ format.

The final call, from the startup code, is to the pre-main routine _main which initialises the standard I/O library prior to calling the main routine. If your program uses no standard I/O you may wish to declare your main function as _main to ensure that no superfluous library routines are drawn in.

Desk Accessory - suffix 'acc'

The desk accessory family of stubs perform the minimal initialisation required by desk accessories. The only 'normal' operations performed are the initialisation for the clock function and the reading of the OS information. The stack space for a desk accessory is fully contained within the startup code, so if a different stack size is required this must be done by modifying the stack size in the startup and reassembling it, the relevant line is:

```
base        ds.b        256
```

which is normally near to the last line in `c.s`.

These stubs perform no command line parsing, no environment setup, no standard file opening and *no* dynamic memory allocator initialisation. This last point is so that all DAs are forced to conform to the requirement that a DA may not legally use GEMDOS Malloc.

In order to circumvent the malloc problem you may add static arrays to the library free memory list using the _addheap function. The function has the prototype:

```
void _addheap(void * base,size_t length);
```

Calling this function with the base of a static array and and a length count adds those bytes to the library heap. For example:

```
int main(void)
{
        static long heap[100];

        _addheap(heap,sizeof(heap));
        ...
}
```

Note that the length of the heap *must* be a longword multiple. Also be aware that if you never call this function and your program calls *any* function which must malloc() memory (e.g. fopen()) then all such calls *will* fail.

Because desk accessories are always GEM programs they must always be linked with the GEM library, additionally the DA startup is specified using the -ta option.

Auto-detecting - suffix 'aut'

This family of stubs allow a program to detect how it has been run, from the auto-folder, as a desk accessory, as a GEM program or as a TOS program. An external variable, int _XMODE, is made available so that the program may perform the requisite initialisation:

0	Standard GEM program
1	Standard TOS program
2	GEM DA
3	Auto folder TOS program

For the GEM/TOS modes (0, 1 and 3) the startup code performs otherwise as described for a normal GEMDOS program. For the DA mode, the stack issue recurs, in this instance the startup code re-uses the space taken by the startup code as the stack for the desk accessory. This gives approximately 750 bytes of stack space, which may be increased by inserting NOPs, immediately before the lsada2 label. Other than this oddity, the DA startup sequence is identical to that described previously.

Resident - suffix 'res'

The resident stubs provide a facility for generating programs which are both residentable and re-entrant. This is used, for example, by LC2 so that it may be multiply re-executed from within the integrated environment. The resident stubs place no special restrictions on the programmer other than that all accesses to the data must be referenced through A4, i.e. the normal -b1 model. Note that these means there must be *no* far data whatsoever.

In the resident mode the startup code initially detects whether or not the program is being executed in a resident manner, or whether it has been run normally, as if by Pexec(0, ...). A copy of the near data section is then made so that any relocation required may be performed on it, prior to initialising A4. Note that in this mode the default locations of any global data variables exported for a symbolic debugger point to the original constant copy of the data and not the copy which is in use.

To use a residentable program from within your own application, it must initially be loaded using Pexec(3, ...) and then shrunk viz:

```
#include <osbind.h>
#include <basepage.h>

bp=(BASEPAGE *)Pexec(3,prog,"","");
Mshrink(bp,bp->p_tlen+bp->p_dlen+0x100);
Mfree(bp->p_env);    /* kill the environment */
```

To execute this program, another basepage must be allocated for the active copy, and then executed:

```
BASEPAGE *ap=(BASEPAGE *)Pexec(5,NULL,command,env);

ap->p_tbase=bp->p_tbase;
ap->p_tlen=bp->p_tlen;
ap->p_dbase=bp->p_dbase;
ap->p_dlen=bp->p_dlen;
ap->p_bbase=ap+1;
ap->p_blen=(long)ap->p_hitpa-(long)ap->p_lowtpa-sizeof(*ap);
err=Pexec(4,NULL,(char *)ap,NULL);
Mfree(ap->p_env);
Mfree(ap);
```

where command and env give the command line and environment respectively that are to be passed to the child process.

If a residentable program is to be executed normally a standard Pexec(0, ...) suffices. Please note that a residentable program whilst conforming to the normal GEMDOS program load format contains many extensions which are generated by the linker in order to support the resident concept. At this time it is *not* our intention to provide detailed information on this since such information is only useful to the startup stubs (provided) and the linker, CLink.

User supplied stubs

A user specified stub may be passed, by the user, to the compiler driver. Typically this is used when the startup code has been modified for a special purpose, or the stack size of a D.A. has been increased, for example.

The name of the stub is communicated to the driver via the -t= option, in conjunction with the -L options. Hence to compile and link with the stub mystub.o:

```
lc -L -t=mystub.o myprog.c
```

Naming conventions

The naming conventions for the startup stubs follows exactly the same pattern as that for libraries. The first character of a startup stub is always c, followed by:

Letter	Type of library
s	Default short integer (-w)
r	Register parameter passing (-rr)
nb	Non-base relative (-b0)

The final three character suffix, as described above, are as follows:

Suffix	Type of startup stub	Option
acc	Desk accessory	-ta
aut	Auto program type detecting	-td
res	Resident program	-tr
User	User specified stub	-t=

Hence a file called `csrnbacc.o` would be a short integer, register passing non-base relative desk accessory startup stub.

Re-assembling c.s

The startup code is provided so that it may be modified as the user requires. To re-assemble it the command:

```
asm [options] c.s
```

should be used, either from a shell, the desktop or as a Tool/Run Other command from the environment.

The options part dictates what sort of stub is to be produced, all types of stub are controlled using various -d parameters. These are:

AUTO	Auto-detecting (-fd)
DA	Desk accessory (-fa)
GST	GST compatible (-z)
NOBASER	Non-base relative mode (-b0)
REGARGS	Register passing mode (-fr)
RESIDENT	Residentable program (-ft)
SHORTINT	Default short integer mode (-w)

Hence to build the stub described earlier, `csrnbacc.o`, the command used is:

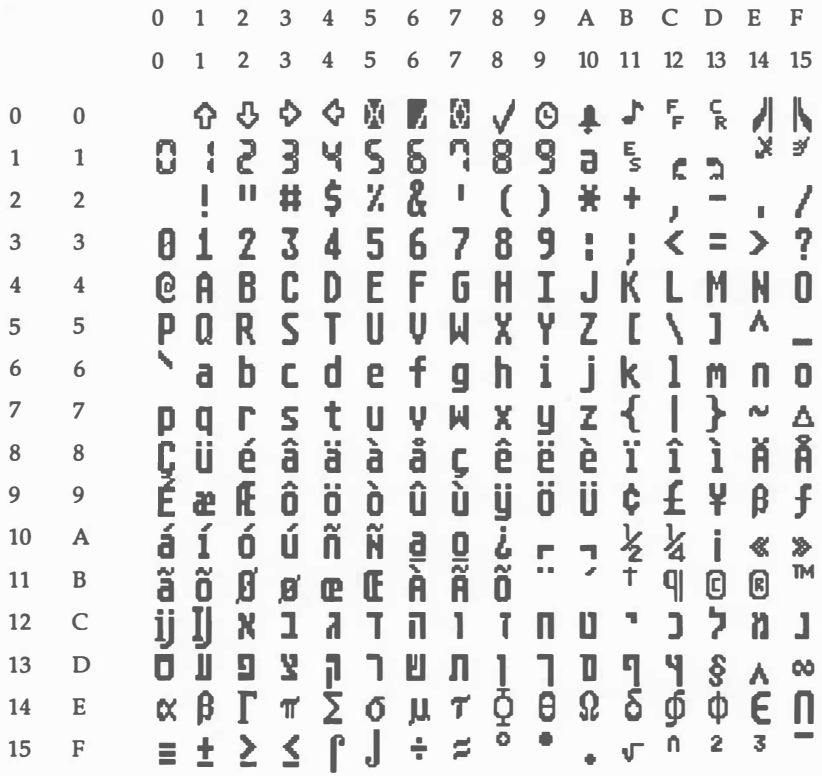
```
asm -dDA -dREGARGS -dNOBASER -dSHORTINT c.s
```

Note that this will in fact build the linkable file into `c.o` rather than `csrnbacc.o`.

Also note that specifying GST does not build a GST compatible linkable file, the `.o` produced by asm must be passed through lc2gst first.

Appendix G
ST ASCII Table

Here is the 8-bit ASCII representation of the ST's character set:



The most significant four bits of the ASCII representation are shown down the left side whereas the least four significant bits are across the top so that, for example:

4C (4*16+12=76 decimal) represents **L**

7B (7*16+11 = 123 decimal) represents **{**

Appendix H
VT52 Screen Codes

When writing to the screen via GEMDOS or the BIOS calls, the screen driver emulates VT52 protocols. The control codes are sent via *escape* sequences, which means an escape character is sent (27 decimal, or $1B) followed by one or more other characters.

ESC A	Cursor up; no effect if at the top line
ESC B	Cursor down; no effect if at the bottom line
ESC C	Cursor right; no effect if on the right hand side
ESC D	Cursor left; no effect if on left hand side
ESC E	Clear screen and home cursor
ESC H	Home cursor
ESC I	Move cursor up one line; if at top scrolls the screen down a line
ESC J	Erase to end of screen, from the cursor position onwards
ESK K	Clear to end of line
ESC L	Insert a line by moving all following lines down. Cursor is positioned at start of the new line
ESC M	Delete a line by moving all following lines up
ESC Y	Position cursor; should be followed by two characters, the first being the Y position, the second the X. Row and column numbering starts at (32, 32) which is the top left
ESC b	Foreground colour; should be followed by a character to determine the colour, of which the four lowest bits are used
ESC c	Background colour; similar to above
ESC d	Erase from beginning of display to the cursor position
ESC e	Enable cursor
ESC f	Disable cursor
ESC j	Save the current cursor position
ESC k	Restore a cursor position saved using ESC j; note that this is not supported on the original 1.0 ROMs
ESC l	Erase a line and put cursor at start of line
ESC o	Erase from start of line to cursor position
ESC p	Inverse video on
ESC q	Inverse video off
ESC v	Wrap around at end of line on
ESC w	Wrap around at end of line off

For instance these codes may be used in a normal printf statement to print information one after another, on the same line, e.g.

```
printf("\033IProcessing %s\033K\n",file);
```

Appendix I
Bibliography

C Programming

Advanced C **Schildt, Herbert (1986)**

ISBN 0-07-881208-9, McGraw-Hill, Berkeley, CA 94710, USA.

Advanced C Techniques & Applications
Sobelman, Gerald E. and David E. Krekelberg (1985)

ISBN 0-88022-162-3, QUE Corporation, Indianapolis, IN 46250, USA.

Advanced C: Food For The Educated Palate **Gehani, Narain (1985)**

ISBN 0-88175-078-6, Computer Science Press, Rockville, MD 20850, USA.

ANSI C: A Lexical Guide **Mark Williams Company (1988)**

ISBN 0-13-037814-3, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, USA.

ANSI X3.159-1989 - Programming Language - C **ANSI (1990)**

American National Standards Institute, 1430 Broadway, New York 10018, USA.

C For Beginners **Sinclair, Ian (1986)**

ISBN 0-86161-206-X, Melbourne House Publishers Ltd., 60 High Street, Kingston-Upon-Thames, Surrey KT1 4DB, U.K.

C Programming Guide **Purdum, Jack (1983)**

ISBN 0-88022-022-8, QUE Corporation, Indianapolis, IN 46250, USA.

C Self-Study Guide **Purdum, Jack (1985)**

ISBN 0-88022-149-6, QUE Corporation, Indianapolis, IN 46250, USA.

C Wizard's Programming Reference **Schwaderer, W. David (1985)**

ISBN 0-471-82641-3, Wiley Press, New York, NY 10158, USA.

C: A Reference Manual, 2nd Edition
Harbison, Samuel P. and Guy L. Steele Jr (1984)

ISBN 0-13-109802-0, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, USA.

Common C Functions **Brand, Kim Jon (1985)**

ISBN 0-88022-069-4, QUE Corporation, Indianapolis, IN 46250, USA.

Debugging C **Ward, Robert (1986)**

ISBN 0-88022-261-1, QUE Corporation, Indianapolis, IN 46250, USA.

Efficient C **Plum, Thomas and Jim Brodie (1985)**

ISBN 0-911537-05-8, Plum Hall, Cardiff, NJ 08232, USA.

Going from BASIC to C **Traister, Robert J. (1985)**

ISBN 0-13-357799-6, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, USA.

Introducing C **Allen, Boris (1986)**

ISBN 0-00383-105-1,William Collins Sons & Co. Ltd., 8 Grafton Street, London W1X 3LA, U.K.

Learning to Program in C **Plum, Thomas (1983)**

ISBN 0-911537-00-7, Plum Hall, Cardiff, NJ 08232, USA.

Microsoft C Run-Time Library **Jamsa, Kris (1985)**

ISBN 1-55615-227-2, Microsoft Press, Bellevue, WA 98009, USA.

Programming In C For The Microcomputer User
Traister, Robert J. (1984)

ISBN 0-13-729641-X, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, USA.

Software Engineering in C
Darnell, Peter A. and Philip E. Margolis (1988)

ISBN 0-387-96574-2, Springer-Verlag, 175 Fifth Avenue, New York, NY 10010, USA.

Standard C **Plauger, P. L. and J. Brodie (1989)**

ISBN 1-55615-158-6, Microsoft Press, 16011 NE 36th Way, Box 97017, Redmond, Washington, 98073-9717, USA..

The C Compendium **Lawrence, David and Mark England (1985)**

ISBN 0-946498-86-6, Sunshine Books, 12-13 Little Newport Street, London WC2H 7PP, U.K.

The C Library **Jamsa, Kris (1985)**

ISBN 0-07-881110-4, McGraw-Hill, Berkeley, CA 94710, USA.

The C Programmer's Handbook **Bolsky, M.I. (1985)**

ISBN 0-13-110073-4, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, USA.

The C Programmer's Handbook **Hogan, Thom (1984)**

ISBN 0-89303-365-0, Brady Communications Company, Inc., Bowie, MD 20715, USA.

The C Programming Language, 2nd Edition
Kernighan, Brian W. and Dennis M. Ritchie (1988)

ISBN 0-13-110370-9, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, USA.

The C Toolbox **Hunt, William James (1985)**

ISBN 0-201-11111-X, Addison-Wesley Publishing Company, Reading, MA, USA.

UNIX System V Programmers Reference Manual **AT&T (1987)**

ISBN 0-13-940479-1, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, USA.

Variations In C **Schustack, Steve (1985)**

ISBN 0-914845-48-9, Microsoft Press, Bellevue, WA 98009, USA.

68000

16-Bit Microprocessors **Whitworth, Ian R. (1985)**

ISBN 0-00-383113-2, William Collins Sons & Co. Ltd., 8 Grafton Street, London W1X 3LA, U.K.

68000 Assembly Language Programming 2nd Edition
Kane, G., D.Hawkins and L.Leventhal (1987)

ISBN 0-07-881232-1, Osborne/McGraw-Hill, 2600 Tenth Street, Berkely, CA 94710, USA.

68000 Machine Code Programming **Barrow, David (1985)**

ISBN 0-00-383163-9, William Collins Sons & Co.Ltd., 8 Grafton Street, London W1X 3LA, U.K.

68000, 68010, 68020 PrimerKelly-Bootle, Stan and Bob Fowler (1985)

ISBN 067-22405-4, Howard W.Sams & Co., 4300 W.62nd Street, Indianapolis, IN 46268, USA.

M68000 Family Programmer's Reference Manual
Motorola Inc. (1989)

Motorola Semiconductor Products Inc., PO Box 20912 Phoenix, AZ 85036, USA.

Mastering The 68000 Microprocessor **Robinson, Phillip R. (1985)**

ISBN 0-8306-1886-4, Tab Books Inc., Blue Ridge Summit, PA 17214, USA.

Microprocessor Systems: A 16-Bit ApproachEccles, William J. (1985)

ISBN 0-201-11985-4, Addison-Wesley Publishing Company, Reading, MA, USA.

Programming the 68000 **Williams, Steve (1985)**

ISBN 0-89588-133-0, SYBEX Inc., 2021 Challenger Drive #100, Alameda, CA 94501, USA.

The MC68000 User's Manual 7th Edition **Motorola Inc. (1989)**

ISBN 0-13-567074-8, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, USA.

The MC68020 User's Manual 2nd Edition **Motorola Inc. (1985)**

ISBN 0-13-566878-6, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, USA.

The MC68030 User's Manual **Motorola Inc. (1987)**

Motorola Semiconductor Products Inc., PO Box 20912 Phoenix, AZ 85036, USA.

The MC68881/MC68882 User's Manual **Motorola Inc. (1987)**

ISBN 0-13-566936-7, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, USA.

Algorithms & Data Structures

Algorithms in C **Sedgewick, Robert (1990)**

ISBN 0-201-51425-7, Addison-Wesley Publishing Company, Reading, MA, USA.

C Chest and Other C Treasures **Holub, Allen I. (1987)**

ISBN 0-934375-40-2, M & T Books, 501 Galveston Drive, Redwood City, CA 94063, USA.

Compilers: Principles, Techniques and Tools
Aho, Alfred V, Ravi Sethi and Jeffrey D. Ullman (1986)

ISBN 0-201-10194-7, Addison-Wesley Publishing Company, Reading, MA, USA.

Data Handling Utilities In C
Radcliffe, Robert A. and Thomas J. Raab (1986)

ISBN 0-89588-304-X, Sybex, Berkeley, CA 94710, USA.

Data Structures and Algorithms
Aho, Alfred V, John E. Hopcroft and Jeffrey D. Ullman (1983)

ISBN 0-201-00023-7, Addison-Wesley Publishing Company, Reading, MA, USA.

Fundamental Algorithms **Knuth, Donald E. (1973)**

ISBN 0-201-03809-9, Addison-Wesley Publishing Company, Reading, MA, USA.

Seminumerical Algorithms **Knuth, Donald E. (1981)**

ISBN 0-201-03822-9, Addison-Wesley Publishing Company, Reading, MA, USA.

Sorting and Searching **Knuth, Donald E. (1973)**

ISBN 0-201-03803-X, Addison-Wesley Publishing Company, Reading, MA, USA.

ST Specific

A Hitchhikers Guide to the BIOS **Atari Corp (1986)**

Atari Corp, 1196 Borregas Avenue, Sunnyvale, CA 94086, USA.

Atari GEMDOS Reference Manual **Atari Corp (1986)**

Atari Corp, 1196 Borregas Avenue, Sunnyvale, CA 94086, USA.

Atari ST Internals 3rd Edition
Brückmann, Rolf, Lothar Englisch and Klaus Gerits (1988)

ISBN 0-916439-46-1, Data Becker GmbH, Merowingerstraße 30, 4000 Düsseldorf, West Germany.

COMPUTE!'s ST Applications Guide: Programming in C
Field, Simon, Kathleen Mandis and Dave Myers (1986)

ISBN 0-87455-078-5, COMPUTE! Publications, Inc., P.O. Box 5406 Greensboro, NC 27403, USA.

COMPUTE!'s Technical Reference Guide, Atari ST - Volume I: VDI
Sheldon Leeman (1987)

ISBN 0-87455-093-9, COMPUTE! Publications, Inc., P.O. Box 5406 Greensboro, NC 27403, USA.

Concise Atari ST 68000 Programmer's Reference
Katherine D. Peel (1986)

ISBN 1-85181-017-X, Glentop Publishers Ltd., Standfast House, Bath Place, High Street Barnet, Herts EN5 5XE, U.K.

GEM Programmer's Guide, Volume 1: VDI **Digital Research (1987)**

Digital Research Inc., 60 Garden Court, P.O. Box DRI, Monterey, CA 93942, USA.

GEM Programmer's Guide, Volume 2: AES **Digital Research (1987)**

Digital Research Inc., 60 Garden Court, P.O. Box DRI, Monterey, CA 93942, USA.

GEMDOS Extended Argument (ARGV) SpecificationAtari Corp (1989)

Atari Corp, 1196 Borregas Avenue, Sunnyvale, CA 94086, USA.

Professional GEM **Oren, Tim (1985)**

ANTIC Publishing

Programmers Guide to GEM Balma, Phillip and William Fitler (1986)

ISBN 0-89588-297-3, SYBEX Inc., 2344 Sixth Street, Berkeley, CA 94710, USA.

Rainbow TOS Release Notes **Atari Corp (1989)**

Atari Corp, 1196 Borregas Avenue, Sunnyvale, CA 94086, USA.

S.A.L.A.D. - Still Another Line A Document **Atari Corp (1987)**

Atari Corp, 1196 Borregas Avenue, Sunnyvale, CA 94086, USA.

STE TOS Release Notes **Atari Corp (1989)**

Atari Corp, 1196 Borregas Avenue, Sunnyvale, CA 94086, USA.

The Pexec Cookbook **Atari Corp (1989)**

Atari Corp, 1196 Borregas Avenue, Sunnyvale, CA 94086, USA.

Appendix J
Technical Support

So that we can maintain the quality of our technical support service we are detailing how to take best advantage of it. These guidelines will make it easier for us to help you, fix bugs as they get reported and save other users from having the same problem. Technical support is available in five ways:

Phone our technical support hour is normally between 3pm and 4pm, though non-European customers' calls will be accepted at other times.

Post if sending a disk, *please* put your name & address on it.

BIX™ our username is (not surprisingly) *hisoft*. Would UK customers please use CIX or more old fashioned methods; it's cheaper for everyone.

CIX™ our username is (still not surprisingly) *hisoft*.

GEnie™ our username is ... OK, you've guessed it!

For bug reports, please always quote the program, computer, version number of the program (the one from the file READ.ME) and the serial number found on your master disk. If your problem is with the compiler please ensure that you tell us *all* compiler options which you used so that we can attempt to duplicate the problem.

If you think you have found a bug, try and create a small program that reproduces the problem. It is always easier for us to answer your questions if you send us a letter and, if the problem is with a particular source file, please enclose a copy *on* disk (which we *will* return).

Remember, technical support is *only* available to registered users i.e. those people that have filled out the registration card enclosed with Lattice C 5 and mailed it to HiSoft.

Upgrades

As with all our products, Lattice C 5 is undergoing continual development and, periodically, new versions become available. We normally make a small charge for upgrades, though if extensive additional documentation is supplied the charge may be higher. All users who return their registration cards will be notified of *major* upgrades.

Suggestions

We welcome any comments or suggestions about our programs and, to ensure we remember them, they should be made in writing.

Common Problems

The following is a list of common mistakes made when starting programming with Lattice C. Please check these carefully against your problem before ringin for technical support.

? Why does the linker complain that symbols beginning with __CX or __AES are undefined and wants me to enter a "DEFINE value"?

• These are symbols in the floating point maths library and/or GEM libraries (respectively), if you have used any floating point maths or GEM calls you must link with the relevant library. This involves selecting Link with floating point or Link with GEM from the options menu if using the integrated compiler, or using the -Lm or -Lg options from the command line.

? My program was nearly finished, but suddenly the compiler has started reporting "Intermediate file error". How can I stop this?

• This almost certainly means that the compiler has run out of space on the device on which it is creating the quad (intermediate) file. Either delete some files from the appropriate disk, or if it is a RAM disk, and you have enough memory left, increase its size.

? Whenever I try to read/write screen images, they always get corrupted. Why?

• The libraries support two modes of file operation, text (the default) and binary. In text mode all CR/LF pairs are considered special and may be removed/added by the library. If using a binary file (such as a screen image) make sure you have selected the binary mode of operation, see the page on fopen() etc. for more details on this.

? My program mysteriously crashes, and I think this is the compiler corrupting the stack. Is this a bug?

• What has probably happened is that you have either allocated too much stack space with stack checks disabled (these are *on* by default), or that you are writing over the end of an array which you have allocated on the stack. This problem can be hard to track down, so always make sure you test your program with stack checks on.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z