Stop on first error/warning -q

Undefined tag warnings -ct

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

Error limit: num -qnume

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

Warning limit: num -qnumw

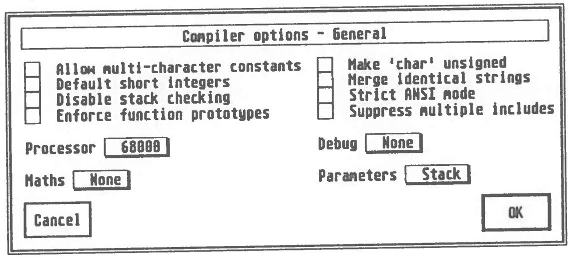
This option controls the number of warnings which the compiler will tolerate before quitting. The default is 50.

Warnings:	
D	*no option*
E	**-j**num**e**
W	**-j**num**w**
I	**-j**num**i**

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

Compiler options - General

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



Compiler options - General

- □ Allow multi-character constants
- □ Default short integers
- □ Disable stack checking
- □ Enforce function prototypes
- □ Make 'char' unsigned
- □ Merge identical strings
- □ Strict ANSI mode
- □ Suppress multiple includes

Processor 68000

Maths None

Debug None

Parameters Stack

Allow multi-character constants -cm

Default short integers -w

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

Disable stack checking -v

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

Enforce function prototypes -cf

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

Make 'char' unsigned -cu

Merge identical strings -cs

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

Strict ANSI mode -ca

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

Enabling this option has the following effects:

> Disables anonymous unions
>
> Disables zero length arrays within structures (note that zero length arrays are *only* permitted within structures).
>
> Disables `long float` as a synonym for `double`.

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

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

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

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

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

Disables warnings:

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

Enables warning:

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

Promotes the following warnings to errors:

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

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

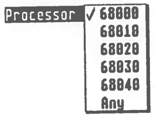
Disallows floating point constant expressions from participating in `case` expressions.

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

Suppress multiple includes -ci

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

Processor -m



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

68000 -m0

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

68010 -m1

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

68020 -m2

Causes the compiler to generate code optimised for the 68020 processor. This code will not run on a 68010 or 68000 although it will run on a 68030 and 68040.

68030 -m3

Causes the compiler to generate code optimised for the 68030 processor. This code will not run on a 68010 or 68000 although it will work on a 68020 and 68040.

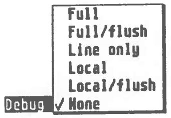
68040 -m4

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

Any -ma

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

Debug -d



```
        Full
        Full/flush
        Line only
        Local
        Local/flush
Debug  ✓None
```

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

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

Full -d4

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

Full/flush -d5

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

Line only -d1

Enables output of the line number/offset table.

Local -d2

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

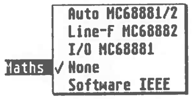
Local/flush -d3

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

None -d0

Disables all debugging information.

Maths -f



```
        Auto MC68881/2
        Line-F MC68882
        I/O MC68881
Maths  ✓None
        Software IEEE
```

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

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

Auto MC68881/2 -fa

Auto-detecting I/O based 68881 emulation routines will be used when this option is specified. The library will check for the presence of an true 68881/68882 coprocessor or an I/O based 68881 (such as Atari's SFP004 or the optional MegaST^{E} coprocessor) and perform floating point arithmetic on chip when possible. If no coprocessor is available then the *Software IEEE* routines will be used.

Line-F MC68882 -f8

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

I/O MC68881 . -fi

I/O based 68881 maths routines will be used when this option is specified. The library assumes the presence of an I/O based 68881 (such as Atari's SFP004 or the optional MegaST^{E} coprocessor) and performs floating point arithmetic on chip.

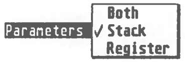
None -f

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

Software IEEE -fl

Standard Lattice IEEE routines linked into the program to perform software emulation of all floating point operations. This code will work on all machines but will not take advantage of a coprocessor if present.

Parameters -r



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

Both -rb

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

Stack -rs

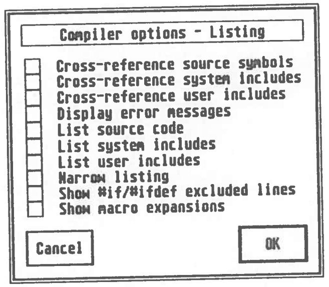
The compiler default, causes the compiler to use standard stack parameters for all subroutine calls. Those functions explicitly declared `__regargs` will use registerised parameter conventions.

Register -rr

Causes the compiler to use registerised parameters for all subroutine calls and entry points. The first two integral and two pointer items will be loaded into `D0-D1/A0-A1` (and the first two real items into `FP0-FP1` if using *Line-F MC68882* (`-f8`)) for the call. Any function without a prototype or explicitly declared `__stdargs` will use the normal stack conventions.

Compiler options - Listing

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



Compiler options - Listing

□ Cross-reference source symbols
□ Cross-reference system includes
□ Cross-reference user includes
□ Display error messages
□ List source code
□ List system includes
□ List user includes
□ Narrow listing
□ Show #if/#ifdef excluded lines
□ Show macro expansions

Cross-reference source symbols -gx

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

Cross-reference system includes -gc

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

Cross-reference user includes -gd

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

Display error messages -go

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

List source code -gs

Enables listing of the input source code.

List system includes -gh

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

List user includes -gi

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

Narrow listing -gn

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

Show #if/#ifdef excluded lines -ge

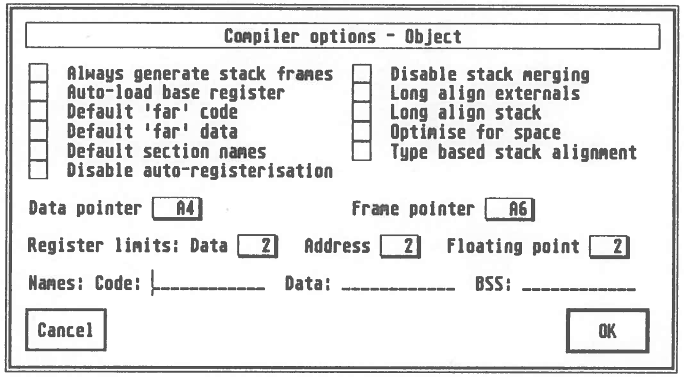
Causes the source listing to display all excluded lines as controlled by `#if` or `#ifdef`. Normally these lines are not displayed.

Show macro expansions -gm

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

Compiler options - Object

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



Compiler options - Object

□ Always generate stack frames
□ Auto-load base register
□ Default 'far' code
□ Default 'far' data
□ Default section names
□ Disable auto-registerisation
□ Disable stack merging
□ Long align externals
□ Long align stack
□ Optimise for space
□ Type based stack alignment

Data pointer A4 Frame pointer A6

Register limits: Data 2 Address 2 Floating point 2

Names: Code: Data: BSS:

Always generate stack frame -mf

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

Auto-load base register -y

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

Default 'far' code -r0

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

Default 'far' data -b0

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

Default section names -s

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

Disable auto-registerisation -mr

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

Disable stack merging -mc

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

Long align externals -cl

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

Long align stack -as

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

Optimise for space -ms

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

Type based stack alignment -aw

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

Code: *codename* -sc=*codename*

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

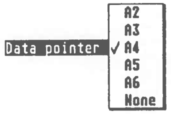
Data: *dataname* -sd=*dataname*

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

BSS: *bssname* -sb=*bssname*

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

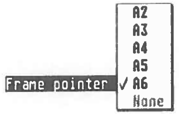
Data pointer: *reg* -b*reg* -bn



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

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

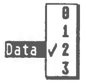
Frame pointer: *reg* -r*reg* -rn



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

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

Register limits - Data: num   *-hd*num


Data
0
1
✓2
3

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

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

Register limits - Address: num   *-ha*num

Address
0
1
✓2

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

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

Register limits - Floating point: num   *-hf*num

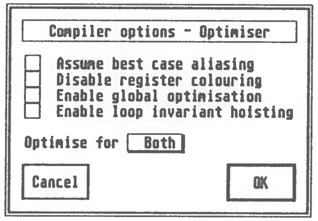
Floating point
0
1
✓2
3

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

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

Compiler options - Optimiser

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


Compiler options - Optimiser

☐ Assume best case aliasing
☐ Disable register colouring
☐ Enable global optimisation
☐ Enable loop invariant hoisting

Optimise for: Both

Assume best case aliasing   *-Oalias*

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

Disable register colouring   *-Onocolor*

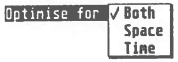
By default the global optimiser examines your code for variables with non-overlapping lifetimes and allows two or more variables to use the same physical register. Using this option disables this behaviour.

Enable global optimisation   *-O*

Enable loop invariant hoisting   *-Oloop*

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

Optimise for   *-O*


Optimise for
✓Both
Space
Time

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

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

Both   -O

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

Space   -Ospace

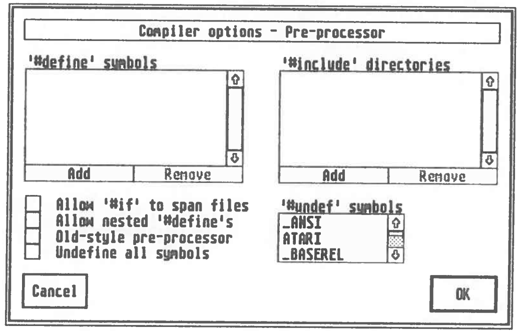
This option performs only very-busy expression hoisting; typically this will result in a decrease in program size.

Time   -Otime

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

Compiler options - Pre-processor

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



Allow #if to span files -cp

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

Allow nested '#define's -cn

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

Old-style pre-processor -co

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

Undefine all symbols -u

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

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

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

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

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

```
#define name value
```

or simply `name` to indicate

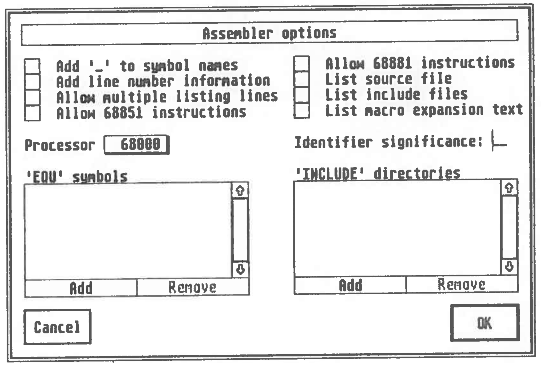
```
#define name
```

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

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

Assembler options

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


Assembler options
□ Add '_' to symbol names
□ Add line number information
□ Allow multiple listing lines
□ Allow 68851 instructions
□ Allow 68881 instructions
□ List source file
□ List include files
□ List macro expansion text
Processor 68000
Identifier significance:
'EQU' symbols
Add
Remove
'INCLUDE' directories
Add
Remove
Cancel
OK

Add '_' to symbol names -u

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

Add line number information -d

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

Allow multiple listing lines -lm

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

Allow 68851 instructions -m9

Allow 68881 instructions -m8

List source file -l

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

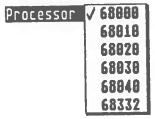
List include files -li

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

List macro expansion text -lx

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

Processor -m



Processor ✓68000
68010
68020
68030
68040
68332

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

68000 -m0

68010 -m1

68020 -m2

68030 -m3

68040 -m4

68332 -m32

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

Identifier significance: *sig* -nsig

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

'EQU' symbols: *name=value* *name* -dname=value -dname

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

```
name EQU  value
```

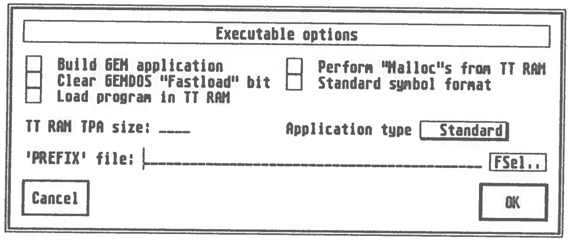
or simply `name` to indicate:

```
name EQU  1
```

'INCLUDE' directories: dir — -idir

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

Executable options



Executable options

□ Build GEM application
□ Clear GEMDOS "Fastload" bit
□ Load program in TT RAM
□ Perform "Malloc"s from TT RAM
□ Standard symbol format

TT RAM TPA size: ____

Application type: Standard

'PREFIX' file: ____

Build GEM application — -Lg

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

Clear GEMDOS "Fastload" bit — NOFASTLOAD

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

Load program in TT RAM — TTLOAD

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

Perform "Malloc"s from TT RAM — TTMALLOC

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

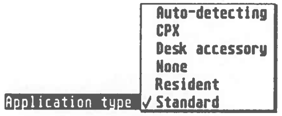
Standard symbol format — DRISYM

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

TT RAM TPA size: size — TPASIZE size

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

Application type



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

Auto-detecting — -ta

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

CPX — -tx

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

Desk accessory — -ta

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

None -t=

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

Resident -tr

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

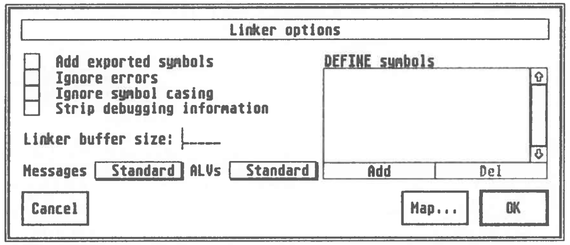
Standard no option

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

'PREFIX' file: file *PREFIX file*

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

Linker options



Linker options

□ Add exported symbols
□ Ignore errors
□ Ignore symbol casing
□ Strip debugging information

Linker buffer size: ____

Messages Standard ALVs Standard

DEFINE symbols

Add exported symbols *ADDSYM*

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

Ignore errors *IGNORE*

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

Ignore symbol casing *NOCASE*

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

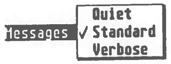
Strip debugging information *NODEBUG*

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

Linker buffer size: size *BUFSIZE size*

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

Messages



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

Quiet QUIET

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

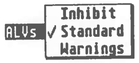
Standard no option

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

Verbose VERBOSE

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

ALVs



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

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

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

Inhibit	**XNOALVS**

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

Standard	no option

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

Warnings	**NOALVS**

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

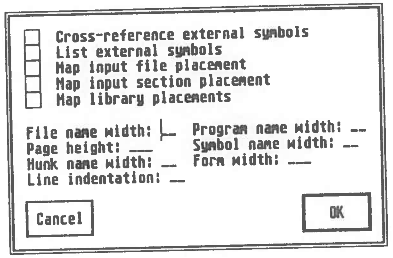
DEFINE symbols: *name=value*	***DEFINE*** *name=value*
name=name	***DEFINE*** *name=name*

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

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

Map...

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



□ Cross-reference external symbols
□ List external symbols
□ Map input file placement
□ Map input section placement
□ Map library placements

File name width: __ Program name width: __
Page height: ___ Symbol name width: __
Hunk name width: __ Form width: ___
Line indentation: __

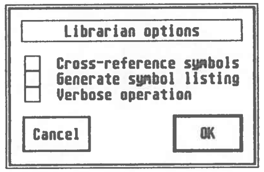
Cross reference external symbols	***MAP...X***
List external symbols	***MAP...S***
Map input file placements	***MAP...F***
Map input section placements	***MAP...H***
Map library file placements	***MAP...L***

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

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

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

Librarian options


Librarian options

☐ Cross-reference symbols
☐ Generate symbol listing
☐ Verbose operation

Cross-reference symbols -x

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

Generate symbol listing -s

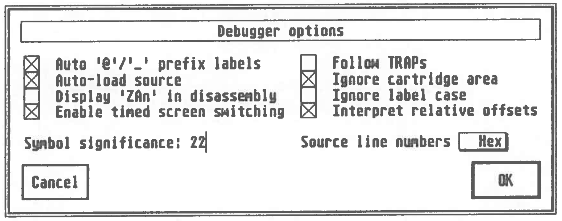
This option causes a librarian listing to be generated and includes a listing of all public symbols defined in the module .

Verbose operation -v

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

Debugger options

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


Debugger options

☒ Auto '@'/'_' prefix labels
☒ Auto-load source
☐ Display 'ZAn' in disassembly
☒ Enable timed screen switching

☐ Follow TRAPs
☒ Ignore cartridge area
☐ Ignore label case
☒ Interpret relative offsets

Symbol significance: 22

Source line numbers Hex

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

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

Auto '@'/'_' prefix labels

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

Auto-load source

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

Display 'ZAn' in disassembly

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

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

Enable timed screen switching

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

Follow TRAPs

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

Ignore cartridge area

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

Ignore label case

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

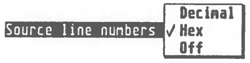
Interpret relative offsets

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

Symbol significance: *sig*

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

Source line numbers

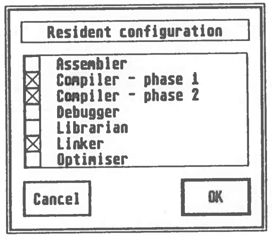


Source line numbers
Decimal
✓ Hex
Off

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

Resident configuration

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



Resident configuration

□ Assembler
☒ Compiler - phase 1
☒ Compiler - phase 2
□ Debugger
□ Librarian
☒ Linker
□ Optimiser

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

LC1, LC2, GO The Compiler

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

New Language Features

ANSI compliance

Extern scoping model

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

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

This change has some surprising side effects:

```
void fn(long);

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

   fn(42);
}
```

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

Flexible keyword ordering

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

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

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

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

Float as single

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

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

Redundant keyword combinations

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

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

Ref/def model

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

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

Restriction of register arrays/aggregates

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

Scoping rules for 'no-linkage' objects

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

Trigraphs

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

Type composition of scoped declarations

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

typedef model

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

```
typedef int t;

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

Valid storage classes of local functions

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

C++ features

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

Comments

C++ style comments are now permitted. The normal //...\n syntax is used.

Ellipsis

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

Anonymous unions

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

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

Floating point __asm support

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

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

__interrupt keyword

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

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

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

ANSI relaxations

Modifiable lvalues

The construct:

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

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

Signed and sized bit fields

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

Zero length arrays

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

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

Listing control directives

#pragma eject

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

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

```
#eject
```

#pragma space *lines*

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

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

```
#space ...
```

#pragma title *title*

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

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

```
#title ...
```

Error control directives

#pragma error *num*

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

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

#pragma ignore *num*

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

#pragma warning *num*

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

New error/warning messages

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

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

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

There are several new error/warning messages:

register __fpX requires -f8 switch on LC1

A function of the form:

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

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

146 (W) long case value in short switch

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

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

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

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

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

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

151 (W) use of ANSI flexible keyword ordering

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

152 (W) cannot define function via typedef name

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

153 (W) use of string constant concatenation

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

159 (W) use of unary minus on unsigned value

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

161 (W) no prototype at definition of public function

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

162 (W) non-ANSI use of ellipsis punctuator

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

```
void fn(const char *s …);
```

This warning is promoted to an error in ANSI mode.

166 unbalanced comment

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

167 (W) nested comment detected

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

170 (W) C++-style comment detected

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

171 (W) implicit cast of integral argument

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

Pre-processor symbols

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

Optional definitions

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

Static definitions

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

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

Permanent definitions

Name	Value	Meaning
__DATE__	"date"	Date on which compilation was started
__FILE__	"name"	Name of main file which is being compiled
__LINE__	n	Current line which is being translated
__REVISION__	50	Current minor version number.
__STDC__	1	ANSI operation mode
__TIME__	"time"	Time at which compilation was started
__VERSION__	5	Current major version number.

Changes to the run-time model

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

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

Register passing mode

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

__saveds and stack checks

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

A2 as a register variable

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

__asm functions

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

```
__asm __stdargs fn(…);
```

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

Signed and sized bit fields

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

LC.TTP

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

New LC.TTP driver options

-+ list the options passed to each phase of the compiler; prior to starting compilation the LC.TTP lists the options which it is going to pass to each phase of the compiler.

-E automatically invoke $EDITOR on error. When this option is used, the -j option is automatically turned on (to generate an error file) and the editor given by the EDITOR environment variable started with the command line:

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

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

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

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

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

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

-j generate error file. By default any listing file is sent to quad file directory, but with the extension .ERR, this may be changed by specifying the option as -j=*filename*.

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

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

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

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

c This invokes the NOCASE option of the linker. It forces case-insensitive linking.

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

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

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

i This letter directs the linker to ignore errors during linking; it is equivalent to the IGNORE linker keyword.

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

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

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

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

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

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

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

For example, -Lm will search lcm.lib before lc.lib, and -Lvg will search 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.

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

-o The exact meaning of -o has changed such that it is now the name of the compilation output file, i.e. if pre-processing or precompiling (processes which do not require LC2) then the output filename is now specified by -o, rather than -q as in pre 5.50 releases.

-N no compile, link named files only. This is useful to allow linking of all files on the command line with no compilation taking place what-so-ever.

-S This option specifies the stack size for any or all compiler phases. Because the compiler uses some recursive algorithms, very complex expressions may cause it to run out of stack space. If this happens, you can increase the stack beyond its 16K default size in the following way:

-S=n Specifies the stack size for phase 1, phase 2, and the optimiser.

-S1=n Specifies the stack size for phase 1.

-S2=n Specifies the stack size for phase 2.

-S0=n Specifies the stack size for the optimiser.

The value n in the preceding list is the number of bytes in the stack. For example, you can specify 16 kilobytes as 16384, 16k or 16K.

-tx use .CPX startup stub

-tx=y use .CPX startup stub, specifying file y as the CPX header to be prepended to the executable.

-Y syntax check only (-y to lc1.ttp).

-z2 generate DRI format object code

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

New compiler options

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

Assume best case aliasing *-Oalias*

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

Pre-processor expansion buffer: size *-zsize*

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

Ignore symbol casing *NOCASE*

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

Linker

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

The new (or modified options are):

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

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

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

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

CLINKWITH; the CLink environment variable

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

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

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

```
TTLOAD
TTMALLOC
```

programs will automatically be linked to go into TT RAM.

ASM
The Assembler

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

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

Basic concepts

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

Source format

Each assembly language source line has the following format:

```
label      operation  operands      comment
```

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

The four fields of the source line are described below:

Label

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

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

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

Operation

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

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

Operands

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

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

Number representations

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

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

An *operator* is one of the following:

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

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

```
ABC+DEF*-PDQ
```

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

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

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

Comment

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

Addressing modes

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

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

where:

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

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

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

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

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

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

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

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

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

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

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

Using the assembler from the command line

The assembler can be run via the following command:

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

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

>listfile

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

options

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

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

-d*symbol*

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

```
symbol    EQU        1
```

-d*symbol*=*value*

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

```
symbol    EQU        value
```

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

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

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

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

x List the expansion text for macros.

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

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

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

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

3 Used for 68030 target. Provides warning if you attempt to use 68000/010/020/040/332 only instructions.

4 Used for 68010 target. Provides warning if you attempt to use 68000/010/020/030/332 only instructions.

32 Used for 68332 target. Provides warning if you attempt to use 68000/010/020/030/040 only instructions.

8 Used for 68881/68882 target.

9 Used for 68851 target.

-nsig Sets the significance of symbols to sig characters. If no size is specified, this option defaults to using 8 character significance.

-opfx Specifies that the output filename (the .O file). If a directory name is specified the output name is formed by prefixing the input filename (the .S file which is being assembled) with pfx. Any drive or directory prefixes originally attached to the input filename are discarded before the new prefix is added. No intervening blanks are permitted in the string following the -o. Note that if a directory name is to be specified as a prefix, a trailing backslash *must* be supplied.

-u This option automatically prefixes all external references with an underline (_). If references to C labels have already been prefixed with an underline, the option is not needed.

-w This option works like the option -dSHORTINT.

filename

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

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

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

Assembler directives

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

COMM symbol,size

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

CNOP offset,alignment

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

```
        cnop        1,4
```

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

```
        cnop        0,2
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

END

Signifies the end of program source.

ENDM

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

label EQU expression

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

IDNT string

Currently ignored, provided for compatibility only.

INCBIN filename

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

INCLUDE filename

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

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

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

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

LIST

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

[label] MACRO

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

MEXIT

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

NARG

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

NOLIST

Switches the assembly listing off.

OFFSET [expression]

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

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

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

OPSYN name,opcode

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

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

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

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

PAGE

Currently ignored, provided for compatibility only.

RORG expression

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

SECTION name[,type]

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

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

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

label SET expression

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

TTL string

Currently ignored, provided for compatibility only.

XDEF symbol[,symbol...]

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

XREF symbol[,symbol...]

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

Conditional assembly

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

IF 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. ELSEIF is accepted as a synonym for the ELSE directive.

ENDC

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

Macro definition

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

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

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

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

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

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

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

```
        arg[=default]
```

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

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

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

The following example illustrates macro definition using the second style:

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

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

```
        MINWORD   ,d2
        rts
```

resulting in the instructions,

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

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

Pre-defined macros and synonyms

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

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

```
DATA     MACRO
         CSECT     data,1
         ENDM

EVEN     MACRO
         CNOP      0,2
         ENDM

TEXT     MACRO
         CSECT     TEXT,0
         ENDM

         OPSYN     ELSEIF,ELSE

         OPSYN     EXITM,MEXIT
```

Interfacing C with assembly language

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

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

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

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

Control sections

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

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

Code sections

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

```
SECTION    name
CSECT      name
```

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

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

```
        XDEF        newtable
```

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

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

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

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

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

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

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

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

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

```
        JSR         _function(PC)
```

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

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

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

```
        XREF        _cfunc

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

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

Data sections

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

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

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

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

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

```
        CSECT     asmdata,data
        XDEF      _entrynum

_entrynum DC.W    15
        END
```

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

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

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

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

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

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

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

```
          XREF      _otherdata
globl     DC.W      42
```

The move instructions will actually be assembled as

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

because the symbols were declared in a near data section.

BSS and offset sections

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

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

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

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

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

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

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

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

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

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

Function Entry Rules

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

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

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

Standard arguments

This is the default method of parameter passing where all function arguments are placed on the stack immediately before the return address. The __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.

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

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

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

The __asm keyword

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

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

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

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

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

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

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

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

Function exit rules

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

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

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

If inline floating point is being used (-f8) then register FP0 is used for floating point returns, viz:

Return Data	Bits	Asm Syntax	Meaning
float	96	FP0	All of FP0
double	96	FP0	All of FP0

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 callingCC (the C and assembly object modules must be linked with the startup code and appropriate libraries). It illustrates many of the points made previously and can be used as a basis for your own function calls.

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

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

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

void
main(void)
{
   unsigned int result;

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

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

```
          CSECT       text,code        control section
          XDEF        _hash,@hash      declarations
          XREF        _maxhash         imported global

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

          END
```

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

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

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

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

```
          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

1 invalid opcode

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

2 unrecognized opcode

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

3 data generation must occur in reloc section

A data generation operation other than DS appeared in an OFFSET section.

4 invalid operands for this opcode

This error can be caused by invalid addressing modes, data size, macro parameters etc.

5 (W) **label ignored <label>**

The label before a directive, such as a conditional, is not a recognised syntax and has been ignored.

6 **must occur inside section**

A data generation directive was used outside a control section.

7 **invalid symbol**

A symbol containing an illegal character or characters was declared.

8 **public symbol not defined <name>**

The program source contained an XDEF directive of a symbol which was not defined in the program.

9 **cannot define absolute public symbol**

10 **external symbol redefined <name>**

11 **invalid expression**

An OFFSET or IF directive contained an invalid expression. This error can also be caused by an expression containing a divide or modulo by zero.

12 **missing label**

An EQU or SET directive was encountered with no corresponding label.

13 **duplicate label <label>**

More than one definition of the same label was encountered.

14 **not inside scope of IF directive**

An ELSE directive was found which did not lie within a conditional control block.

15 **invalid origin**

An assembly directive causing incorrect data alignment or origin was found.

16 **constant size not same as relocation size**

A reference to a relative symbol conflicts with the byte size of relocation information for the current control section.

17 **invalid string**

A define constant or condition directive contained an invalid string.

18 (W) **extraneous data on input line**

A valid source line was followed by invalid text, which was ignored. This can be caused by providing too many parameters for an assembler directive.

19 **duplicate section name**

A section name was re-used illegally. This error can also be caused by an invalid OPSYN directive.

20 **ELSE/ENDIF not found**

An unterminated IFcc directive was encountered.

21 **label offset different in pass 2**

A phasing error caused by different code being generated on the first and second pass.

22 **macro argument too large**

A macro invocation was encountered with an argument string which was too long.

23 **missing macro definition**

The definition of a macro could not be found.

24 **illegal macro definition**

The syntax of a macro definition was incorrect.

25 **duplicate macro definition**

A macro was defined more than once.

26 **invalid control section parameter**

A CSECT directive with invalid parameters was encountered.

27 **invalid file name**

The filename specified for an INCBIN or INCLUDE directive was not valid.

28 **maximum include file nesting exceeded**

The INCLUDE directive has nested files too deeply. This is caused by included files referencing other files to a number of levels.

29 **file not found**

The file specified by an INCLUDE directive could not be found.

30 **invalid repeat count**

31 **macro substitution line overflow**

The substitution of macro arguments caused the line to overflow.

32 **immediate data out of range**

An arithmetic or logical operation was specified with an out of range immediate value.

33 **invalid effective address for opcode**

An attempt is being made to use an addressing mode not supported by the current instruction.

34 **invalid instruction size**

An attempt is being made to use an instruction size not supported by the current instruction.

35 **target out of range**

A reference to a label which is out of range has been encountered.

36 **value out of range for addressing mode**

An out of range value was used in an addressing mode.

37 **input line buffer overflow**

A source line has exceeded the maximum length.

38 **long branch to XDEF not supported by linker**

The Lattice linkable object file format does not support branches to external labels using a long-word offset.

39 **macro buffer overflow**

A macro definition was too long.

40 **ENDM/MEXIT not inside macro definition**

An ENDM or MEXIT assembly directive was encountered outside a macro definition.

41 **target not in current section**

42 (W) **END directive assumed**

This warning notifies you that there was no explicit END directive in the source file being assembled.

43 **invalid relocation type/size combination**

The specified relocation type and relocation data size specified in a CSECT directive are not available.

44 **unknown segment type**

The type specifier for a SECTION or CSECT directive was other than CODE, DATA or BSS.

45 **numeric value out of range**

A value in an expression overflowed the allowable range.

46 **opcode generated for <processor>**

An opcode only permitted for the indicated processors was generated.

47 **unrecognized expression**

48 **syntax error**

49 **invalid operation for relocatable data**

50 **undefined symbol**

51 **invalid opcode parameter reference**

52 **location counter not defined**

DERCS
The Resource Decompiler

Introduction

DERCS is utility for turning a resource file created using WERCS into a set of initialised data structures (`OBJECT`, `BITBLK`, `ICONBLK` etc.) which may subsequently be compiled to give a resource file embedded in a program.

This is advantageous when creating desk accessories (since desk accessories should not call `rsrc_load()`) and *essential* for writing control panel extensions (CPXs).

DERCS supports generation of both C and assembly langauge. Unfortunately the initialisation support available from other languages is not sufficiently rich to allow the representation of general resource files and so if a language of anything other than C is selected DERCS will generate an assembly language file.

Running DERCS

DERCS is run using a command line of the form:

```
dercs [-options] filename [filename]
```

The options are denoted by a - sign then a character *before* the filename. The options recognised are:

`-a` don't generate ANSI style function definitions. Normally DERCS generates any functions which it requires using the ANSI prototype syntax, viz:

```
void rsrc_init(void);
```

specifying this option causes it to use the old K&R syntax for function definitions. Obviously this flag has no effect for assembly language.

`-cx` specify casing is to be peformed according to *x* rather than the value set in the .HRD file; the values used for *x* are identical to those used by WERCS:

0	mixed
1	upper
2	lower

`-dx` specify *x* as name of the section data is to be placed in. This option allows you to configure the name of the section which DERCS generates when outputting the data section information for assembly language. If none is specified it defaults to DATA. Note that Lattice C ASM users may like to change this to '`__MERGED,data`'. Obviously this flag has no effect for C.

`-f` Suppress output of tree fixup code. Normally DERCS generates a function which you may call which 'fixes up' the resource trees in your file (i.e. the conversion from character to pixel co-ordinates). Specifying this options suppresses this behaviour. You can use this option if you want to fix up trees yourself, or if you are using more than one resource file.

`-h` write only a header file. This option writes only a header file in the desired langauge. Note that with careful use of the `-l` option this allows the generation of resource file constants for more than one language (e.g. in a mixed assembler/C project).

`-lx` use language specified by language number *x*; the values used for *x* are identical to those used by WERCS:

1	C
2	Pascal
4	Modula-2
8	FORTRAN
16	Assembler
32	BASIC

Note that for DERCS values 2 through 32 *all* generate assembly language.

`-px` use *x* as the prefix for automatically generated names. When DERCS is generating some of the more complex AES object structures it has to generate its own names for items which are unnamed in the resource file. If you are merging two or more DERCS'd resource files then this option can be used to ensure that no naming clashes occur.

`-v` suppress output of `section` directives or `include` directives. This means that DERCS will not output any of the 'padding' which it normally generates.

`-xic` generate pre-fixed resource file. This option is designed for building CPXs and fixes up all characters based on an 8x16 character. If *`ic`* is supplied, this should be the name of a free image in the resource file which is to be extracted into a .ICN file ready to be passed to CPREFIX.

The file specified first on the command line is then decompiled into either a C source file (if C was selected as the language in WERCS), or into assembler otherwise.

An optional second file name may be supplied to indicate a name for the output file.

Programming with DERCS

Programming using a resource file passed through DERCS is not dissimilar to using an ordinary resource file, the main difference is that `rsrc_gaddr()` is never used. Instead, DERCS builds all of the necessary resource information into a C file with the extension .C. This file also contains a special resource intialisation routine `rsrc_init()` (this routine actually performs the normal character to pixel coordinate transformations made by `rsrc_load()`).

Consider the following fragment written without DERCS:

```
#include <aes.h>
#include "resource.h"

int main(void)
{
   OBJECT *myobj;

   appl_init();
   rsrc_load("resource.rsc");
```

```
  rsrc_gaddr(TREE, MYOBJ. &myobj);
  objc_draw(myobj, ROOT, …);
  …
  rsrc_free();
  appl_exit();
}
```

Now assume that the command:

```
dercs resource.rsc
```

had been issued, generating the files resource.c and resource.h, then the same program could be written with DERCS:

```
#include <aes.h>
#include "resource.h"
#include "resource.c"

int main(void)
{
  appl_init();
  rsrc_init();
  objc_draw(MYOBJ, ROOT, …);
  …
  appl_exit();
}
```

Note that the `rsrc_load()` call has been replaced by `rsrc_init()`, and that the `rsrc_gaddr()` and `rsrc_free()` have disappeared altogether!

The object `MYOBJ` (which was what you named your object in resource.rsc) which is normally declared as a `#define` constant by WERCS is replaced by an `OBJECT` definition by DERCS, hence the address of the object (which `rsrc_gaddr()` would have obtained) is found simply by naming the object.

CPXBUILD
CPX .HDR Utility

Introduction

CPXBUILD is used to build the .HDR files used as arguments to the linker `PREFIX` option which specify the parameters for a CPX.

Running CPXBUILD

CPXBUILD is run using a command line of the form:

```
cpxbuild [-options] filename
```

The options are denoted by a `-` sign then a character *before* the filename. The options recognised are:

`-b` set boot-init flag; the boot initialisation flag of the CPX is set indicating that the CPX should be called during XControl's initialisation.

`-ccol` set title colour; the colour of the title is set to `col`. The colour numbers used are those used by the AES.

`-did` set CPX-id; the CPX id is set to `id`. If `id` has the special format `'cpid'` then the specifed alphanumeric `cpid` is used as the long-word CPX id.

`-fval` set flags absolutely; the header flags are set to the absolute value `val`; this option is provided to support any future additions which Atari may make to XControl during the lifetime of a CPXBUILD release.

`-iicn` specify an icon file; `icn` specifies a .ICN file which is to be used as the CPX's icon. The .ICN file is generated using DERCS' `-xic` option.

`-ntxt` set icon text; the descriptive text attached to the icon is set to `txt`.

`-pcol` set icon colour; set icon colour; the colour of the title is set to `col`. The colour numbers used are those used by the AES.

`-r` set resident flag; the resident flag of the CPX is set indicating that the CPX should be made resident at the time of XControl's startup.

`-s` set set-only flag; the set-only flag for the CPX is set; this indicates that the CPX performs all its work at boot time and need never be called again.

`-ttxt` set title text; the title text for the CPX is set to `txt`.

`-V` force CPXBUILD to print sign-on and version numbers. Normally CPXBUILD runs silently; this option causes more information to be generated.

`-vvsn` set CPX version; the CPX version number is set to `vsn`. Typically this will have the format `major.minor`.

The `filename` given on the CPXBUILD command line indicates the name of the .HDR file which should be built. Note that the .HDR is *not* automatically supplied; you must specifiy it if required.

Appendix A
Project file syntax

A project file is the file used by the integrated compiler to control the management of multi-module programs. These project (.PRJ) files are in an ASCII format and are compatible with those used by the German Pure-C™ and PKS-Edit™. The syntax may be described by the following grammar:

project:
    (***** | *filename*) { *options* } = { *module* [**(***dependents***)**] }

options:
    .L **[** *linker-options* **]**
    .C **[** *compiler-options* **]**
    .S **[** *assembler-options* **]**
    .A **[** *librarian-options* **]**

module:
    c-source-file **[** *compiler-options* **]**
    assembler-source-file **[** *assembler-options* **]**
    linker-file | *with-file* | *project-file*

c-source-file:
    ***** | *filename* | *filename***.C**

assembler-source-file:
    *filename***.S**

linker-file:
    *filename***.O** | *filename***.LIB**

with-file:
    *filename***.LNK**

project-file:
    *filename***.PRJ**

dependents:
    filename [**,** *dependents*]