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

```
#define __ROL_L_D1_D0    "e3b8"

unsigned long _lrotl(unsigned long,int);

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

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

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

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

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

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

Compiler Operational Errors

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

-i option ignored

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

-r option has been moved to LC1

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

__builtin_fpc requires -f8 switch on LC1

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

Argument to abs must be an integral type

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

Argument to emit must be an integral constant

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

Arguments to max/min must be an integral type

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

Can't create debugger intermediate file

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

Can't define __LINE__ and __FILE__

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

Can't open debugging file

The second phase could not open the debugging symbol file.

Can't open precompiled header file

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

Can't create object file

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

Can't create quad file

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

Can't open file for pre-processor output

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

Can't open prototype file

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

Can't open quad file

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

Can't open source file

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

Can't open symbol file

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

Combined output file name too large

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

Corresponding message not found in lc1.lc

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

Dead assignment eliminated <symbol>

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

End of file on object file

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

Error message too long in lc1.lc

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

Error reading symbol file

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

File name missing

The source file name was not specified.

File name too large

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

File too short

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

Floating point opcode must be a constant

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

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

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

Intermediate file error

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

Invalid -b option

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

Invalid -e option

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

Invalid -f option

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

Invalid -j option. Warning/error not specified

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

Invalid -m option

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

Invalid -r option

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

Invalid -s option

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

Invalid attribute flags for builtin function

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

Invalid intermediate file

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

Invalid option

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

Invalid register specification for getreg

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

Invalid register specification for putreg

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

Invalid symbol definition

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

lc1.lc is corrupt

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

No functions or data defined

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

No register specified for ASM call

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

Not enough memory

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

Parameters beyond file name ignored

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

Reference has overlapping definition <symbol>

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

Same register used twice for parameters

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

Seek error on object file

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

Symbol file corrupted

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

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

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

Unable to find 'lc1.lc'

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

Undefined variable <symbol>

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

Unrecognized -c option

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

Unrecognized -j option

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

Unrecognized -p option

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

Unrecognized -q option

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

Value for putreg must be an integral type

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

Wrong number of parameters for builtin function

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

Syntax errors and warnings

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

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

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

where the message components are:

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

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

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

1 invalid pre-processor command

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

2 unexpected end of file

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

3 file not found <name>

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

4 invalid lexical token

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

5 invalid macro usage

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

6 line buffer overflow

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

7 file stack full

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

8 invalid conversion

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

9 undefined identifier <name>

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

10 invalid subscript expression

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

11 string too large or not terminated

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

12 invalid structure reference

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

13 member name missing

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

14 undefined member <name>

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

15 invalid function call

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

16 invalid function argument

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

17 too many operands

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

18 unresolved operator

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

19 unbalanced parentheses

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

20 invalid constant expression

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

21 illegal use of aggregate

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

22 (W) structure used as function argument

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

23 invalid use of conditional operator

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

24 pointer operand required

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

25 modifiable lvalue required

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

26 arithmetic operand required

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

27 arithmetic or pointer operand required

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

28 missing operand

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

29 invalid pointer operation

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

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

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

31 integral operand required

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

32 invalid conversion specified

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

34 invalid initialiser expression

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

35 closing brace expected

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

36 (W) control cannot reach this statement

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

37 duplicate statement label <label>

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

38 unbalanced braces

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

39 invalid use of keyword <keyword>

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

40 break not inside loop or switch

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

41 case not inside switch

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

42 invalid case expression

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

43 duplicate case value

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

44 continue not inside loop

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

45 default not inside switch

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

46 more than one default

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

47 while missing from do statement

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

48 invalid while expression

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

49 else not associated with if

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

50 label missing from goto

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

51 label name conflict <label>

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

52 invalid if expression

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

53 invalid return expression

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

54 invalid switch expression

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

55 (W) no case values for switch statement

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

56 (W) colon expected

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

57 (W) semi-colon expected

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

58(W) missing parenthesis

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

59 invalid storage class

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

60 incompatible aggregate types

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

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

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

62 structure/union type mismatch

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

63 (W) duplicate declaration of item <name>

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

64 structure contains no members

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

65 invalid function definition

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

66 (W) invalid array limit expression

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

67 illegal object

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

68 illegal object for structure

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

69 structure includes instance of self

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

70 invalid use of structure qualifier

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

71 formal declaration error <name>

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

72 external item attribute mismatch

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

73 (W) declaration expected

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

74 (W) initialiser data truncated

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

75 invalid sizeof expression

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

76 left brace expected

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

77 identifier expected

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

78 undefined statement label <label>

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

79 (W) duplicate enumeration value

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

80 invalid bit field

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

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

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

82 maximum object/storage size exceeded

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

83 (W) reference beyond object size

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

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

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

85 (W) function return value mismatch

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

86 (W) formal definitions conflict with type list

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

87 (W) argument count incorrect

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

88 (W) argument type incorrect

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

89 (W) constant converted to required type

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

90 invalid argument type specifier

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

91 illegal void operand

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

92 (W) statement has no effect

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

93 (W) no reference to identifier <name>

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

94 (W) uninitialised auto variable <name>

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

95 (W) unrecognised #pragma operand

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

99 (W) attempt to change a const lvalue

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

100 (W) no prototype declared for function

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

101 redundant keywords in declaration

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

102 conflicting keywords in declaration

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

103 (W) uninitialised constant <name>

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

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

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

106 postfix expression not allowed on a constant

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

107 too many initialisers

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

109 Invalid use of type name or keyword

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

116 (W) Undefined enum tag <name>

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

117 Enum contains no members

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

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

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

119 Identifiers missing from definition of function <name>

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

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

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

122 (W) Missing ellipsis

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

123 No tag defined for enumeration (cannot construct prototype)

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

124 Debugger symbol table overflow

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

125 Invalid number

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

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

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

127 Operand to # operator must be a macro argument

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

128 #error usage

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

132 (W) Extra tokens after valid preprocessor directive

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

133 Cannot redefine macro <name>

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

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

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

136 Invalid use of register keyword

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

139 (W) Missing #endif

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

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

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

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

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

143 Object has no address

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

144 Combined storage for strings and constants exceeds maximum

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

154 (W) no prototype declared for function pointer

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

155 (W) no statement after label

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

Internal Errors

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

```
CXERR: xx
```

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

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

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

CLink
The Linker

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

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

A simple CLink command line

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

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

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

Concepts

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

ALVs

When CLink is collecting all of the CODE type sections together, if any are more than 32K apart and a 16-bit PC relative access is attempted, rather than simply fail with and out-of-range error message, CLink redirects the access to a JMP to the same location. This jump is known as an *automatic link vector* or *ALV*. Note that this may cause problems if you attempt to access data using PC-relative mode, although this is not recommended anyway since on the 68030 there are separate code and data caches which can cause consistency problems.

Near DATA/BSS

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

Directives

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

Input directives

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

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

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

FROM files

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

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

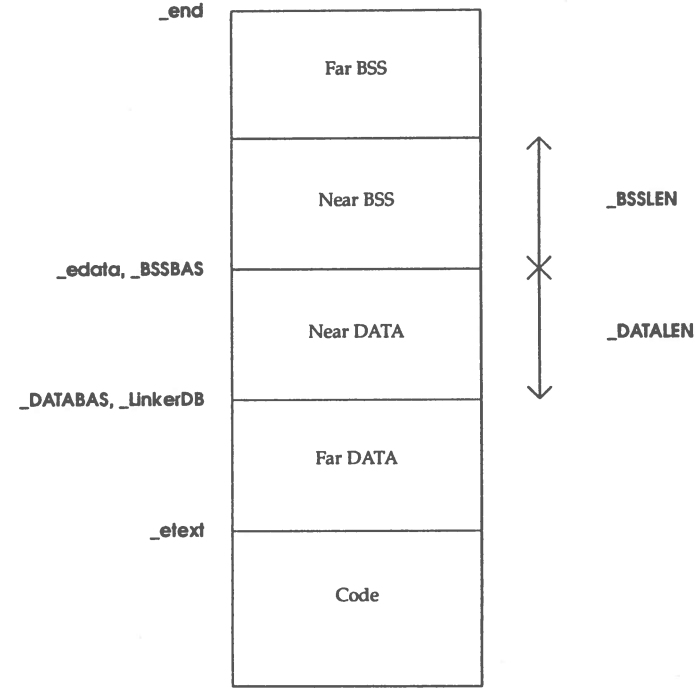
```
FROM a.o+b.o
```


LIB files

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

Output directives

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



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

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

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

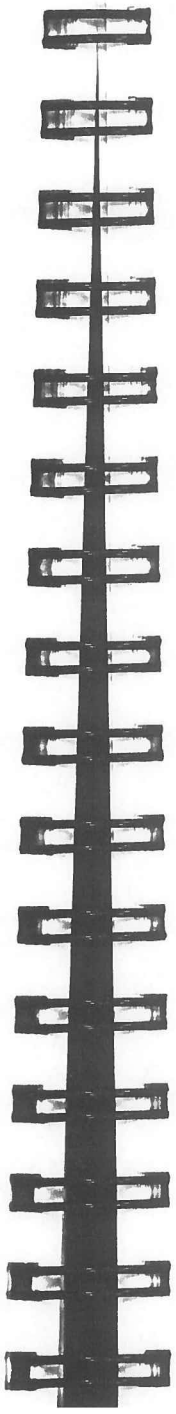
PRELINK Causes CLink to output an object module with references and definitions still intact so that it can be linked later on to produce a final executable file. This is designed for development of large projects where the programmer is only changing a single source module. Note that a prelinked object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

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

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

Pre-linking

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



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

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

Map files

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

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

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

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

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

```
XREF filename
```

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

FWIDTH n — Width of file names (default 16).

HEIGHT n — Lines on a page in map file, 0 indicates no pagination (default 55).

HWIDTH n — Width of hunk names (default 8).

INDENT n — Columns to indent on a line. This is included in width (default 0).

PWIDTH n — Width of program unit names (default 8).

SWIDTH n — Width of symbol names (default 8).

WIDTH n — Sets the maximum line length for the map and cross reference listings. This is useful when sending the output to a device which has different line length requirements. If not specified one width defaults to 80.

'WITH' files

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

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

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

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

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

```
CLINK WITH mywith.lnk
```


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

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

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

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

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

Compiler Options and CLink

The -b1 option

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

The -r1 option

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

The -d options

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

Reserved symbols

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

__RESBASE, __RESLEN — Reserved symbols

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

__LinkerDB — Pointer to static merged data section

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

__BSSBAS, __DATABAS — Base of merged data sections

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

```
extern void *far _BSSBAS,*far _DATABAS;
…
        void *x,*y;

        x=&_BSSBAS;
        y=&_DATABAS;
```

__BSSLEN, __DATALEN — Merged section lengths

The addresses of these names give the length of the respective __MERGED data section in *longwords*. Note that these variables *must* be accessed as longs otherwise the compiler may attempt to relocate them, giving random values as a result.


```
extern void *far _BSSLEN,*far _DATALEN;
…
        long a,b;

        a=(long)&_BSSLEN;
        b=(long)&_DATALEN;
```

__end, __edata, __etext — Last locations in program

These names refer to the last locations in the program. The address of _etext is the first location above the executable program text, that of _edata the first location above the initialised data area and _end the location immediately after the uninitialised data area.

Standard libraries

Because of the large number of options, a wide selection of libraries for use under the various compilation models is supplied. The 'normal' libraries are: c.o, the startup code; lc.lib the standard C library; lcm.lib the standard maths library; lcg.lib the standard GEM library.

All libraries are named according to consistent naming conventions made up of an initial leading letter, either c for a startup stub, or lc for a library, optionally followed by one of the following letters to indicate the type of the library:

Letter	Type of library
m	Library is for floating point maths
g	Library is for GEM support

Note that the standard C library must always be linked *after* the maths or GEM libraries but that the relative order of maths and GEM libraries is unimportant.

Following this prefix, none, one or more of the following letters are used to indicate the options used to compile the library:

Letter	Type of library
s	Default short integer library (-w)
r	Register parameter passing library (-rr)
nb	Non-base relative library (-b0)

The startup stubs may also be followed by an additional suffix; currently the suffices allocated are:

Suffix	Type of startup stub
acc	Desk Accessory
aut	Auto program type detecting
res	Resident program

Hence for instance the file lcsrnb.lib is a short integer, register passing non-base relative C library, whilst lcgr.lib would be a long integer register passing GEM library. Similarly a file called csrnbacc.o would be a short integer, register passing non-base relative Desk Accessory startup stub.

The GST libraries supplied follow identical naming conventions but use a .bin file extension to indicate that they are in the GST format.

CLink Messages

Whilst running CLink may discover things which it needs to bring to your attention. These may either be error messages or observations on the program which is being built.

CLink Warnings/messages

The messages in this section although warnings, will often indicate that the final program will be unusable in the form intended and you should not run it unless you are certain that you understand what you are doing.

Warning MERGED data > 64K, use -b0 on LC

The merged data section has exceeded the limit of 64K. The problem may be rectified by either compiling your program using the -b0 option on lc, or explicitly declaring some of your arrays far.

Warning! Absolute reference to <name> module: <mod> file: <file>

An absolute reference was detected to a merged data item, whilst building a resident load module. This warning will only be given if a reference has been made to the symbol __RESBASE, i.e. the linker is building a resident load module.

Warning: ALVs were generated

This message is generated when the NOALVS option is used, indicating that ALVs were generated. Note that this message will not be issued if the XNOALVS option is used.

Enter a DEFINE value for <name> (default __stub): Undefined symbols... First Referenced

These messages indicate that the linker has encountered a reference to a symbol for which it cannot locate a definition. The second message is issued if the BATCH keyword is specified, whereas the first allows you to specify an alternate name for the reference.

This error may occur because you have called a function using one name, but possibly mis-spelt the definition. Alternatively you may not be linking with the correct libraries, e.g. if the symbols mentioned include names such as _CXA55 then you are not linking with the floating point maths library, similarly, references to variables such as __AESpb indicate that you have used GEM, but not linked with the GEM library.

CLink Errors

These are the errors which may be issued by the linker. In general errors may be ignored by use of the IGNORE keyword to CLink, however programs so produced may not function correctly. The error numbers are broadly divided so that 200-400 may be issued by either pass. 401-500 are issued by pass 1, whilst 501-599 are issued during pass 2.

Note that if a module has been compiled with debugging turned on (-d1 through -d5) then the line number on (or near) to where the problem occurred will also be reported.

200 Out of memory!

The linker does not have enough memory left to successfully complete the link.

300 System error <val> on read

A system error occurred whilst attempting to read from the disk. This should only occur if the disk has been damaged in some way. The value of the error is given by <val>.