

LEA MACRO ASSEMBLER

User's Guide

Revision 1.7, 29th June 2019

License and Disclaimer

LEA Macro Assembler

All versions are copyright © 2019 John Croudy (the author).

<https://github.com/croudyj>

<https://hackaday.io/croudyj>

<http://leo1cpu.puntett.net/>

Permission is hereby granted, free of charge, to any person or organization obtaining a copy of the software and accompanying documentation covered by this license ("the Software") to use, reproduce, display, distribute, execute, and transmit the Software, and to permit third-parties to whom the Software is furnished to do so, all subject to the following:

The copyright notices in the Software and this entire statement, including the above license grant, this restriction and the following disclaimer, must be included in all copies of the Software, in whole or in part. The Software may be used for commercial purposes but it must not be sold. The Software is not fault-tolerant and is not designed or intended for use in hazardous environments requiring fail-safe performance, in which the failure of the Software could lead directly to death, personal injury, or severe physical or environmental damage.

All product names and brands mentioned in the Software are the property of their respective owners and are for identification purposes only. Use of these names and brands does not imply endorsement.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT SHALL THE AUTHOR OR ANYONE DISTRIBUTING THE SOFTWARE BE LIABLE FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF, OR IN CONNECTION WITH, THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Conventions used in this document

- Optional items are enclosed in square brackets []
- Example source code and names of directives are written in a `fixed-width` font.
- When a concept is introduced for the first time, it appears in *italics*.
- The symbol ⚠ denotes a warning about a limitation of the program.
- The symbol 🛠 denotes a helpful tip or piece of advice.

LEA Assembler Overview

LEA (pronounced "lee-ah") is a fully featured macro assembler which can be used to turn assembly language programs into machine code ready to run on a target computer system. It was designed to be extensible and can assemble programs for more than one *target CPU*. The current version is able to assemble programs written for the following two target CPUs:

- **Motorola 68000:** This CPU is automatically selected if the file extension is 68K or X68. For more information on this CPU, please see https://en.wikipedia.org/wiki/Motorola_68000
- **LEO-1:** This CPU is automatically selected if the file extension is LEA. For more information on this CPU, please see <http://leo1cpu.puntett.net/>

More CPU types are planned for future versions, in particular the 6502, Z80 and SC/MP. These are all microprocessors that the author of LEA finds interesting enough to possibly implement a custom CPU for.



LEA is a command-line tool. It does not come with any kind of 'IDE' or other graphical interface. It is run by opening a command window and typing `lea`, or by using a batch file. It is therefore possible to use LEA alone or as part of an automated tool chain.

Compatibility with older assemblers

Over the years, the author has gained familiarity with a number of assemblers, the most recent being EASy68K and before that DevPac ST and DevPac Amiga. These assemblers have influenced the design of LEA to some extent and LEA attempts to behave as much like these older assemblers as possible. However, some significant differences do exist:

- LEA does not restrict white space in expressions and operand lists. This means comments need to start with a semicolon.
- LEA uses 64-bit arithmetic. This may cause some differences if a value would have overflowed past 32-bits.
- LEA does not support IFC, IFNC, IFEQ etc. It uses one unified expression syntax which is a little like an early-80s implementation of structured BASIC.

⚠ Because of these differences, you may find that LEA does not successfully assemble existing 68000 code without errors until the code has been modified.

Overview of operation

Assembly language programs are often written in one or more files which are assembled separately and then linked with a linker to produce the final program. LEA does not include a linking step; it simply assembles a single source file and directly produces machine code output. This source file, often called *Main.ext* (where *ext* is the extension for a particular CPU type) can, of course, include as many other source files as necessary by using the `include` directive. It is common for the main file to include other files such as definitions, macros and so on, then produce the code for the main program, and finally include other parts of the program (subroutines). This makes it possible to split a program into as many pieces as necessary for easier editing. The file extension is also used to decide which target CPU to assemble the code for.

The main file to be assembled is specified on the command line. If the filename is fully-qualified, it can be anywhere in the file system. If not, LEA looks for it in the current folder on the current drive. During assembly, the current folder is changed to the folder where the main file is. All files specified in `include` and `incbin` directives can be either fully-qualified or relative to the current folder, that is, the same folder as the main file.

Source Code File Formats

Internally, LEA works entirely with UTF-8 strings. All text read from input files is assumed to be UTF-8. This means it can read any pure ASCII text file, and it can also read ANSI text files but any bytes with values above 128 will be handled according to UTF-8 conventions rather than as 'codepage' values.

Unicode support

If any non-ASCII codepoints are found in symbol names or comments, LEA will attempt to ensure that those codepoints are transferred through the system unmangled. Thus it should be possible to write programs that have, for example, Asian characters in comments and variable names and the listing should show these comments and names correctly.

Character case

When converting characters to upper or lower case (and when checking strings with case-insensitivity), LEA only considers the Latin alphabetic characters A to Z, and strings stay the same length when the case is changed. This means that, for example, the German character ß (Unicode U+00DF) is not considered to be 'alphabetic' and is ignored by case conversions, as are all the other non-ASCII Unicode codepoints.

BOMs

Byte Order Marks (BOMs) may optionally appear in source files. If no BOM is present, UTF-8 is assumed. If a UTF-16 (BE or LE) BOM is encountered, the file will be loaded and converted internally to UTF-8 for processing. All output files are written as UTF-8 with no BOM.

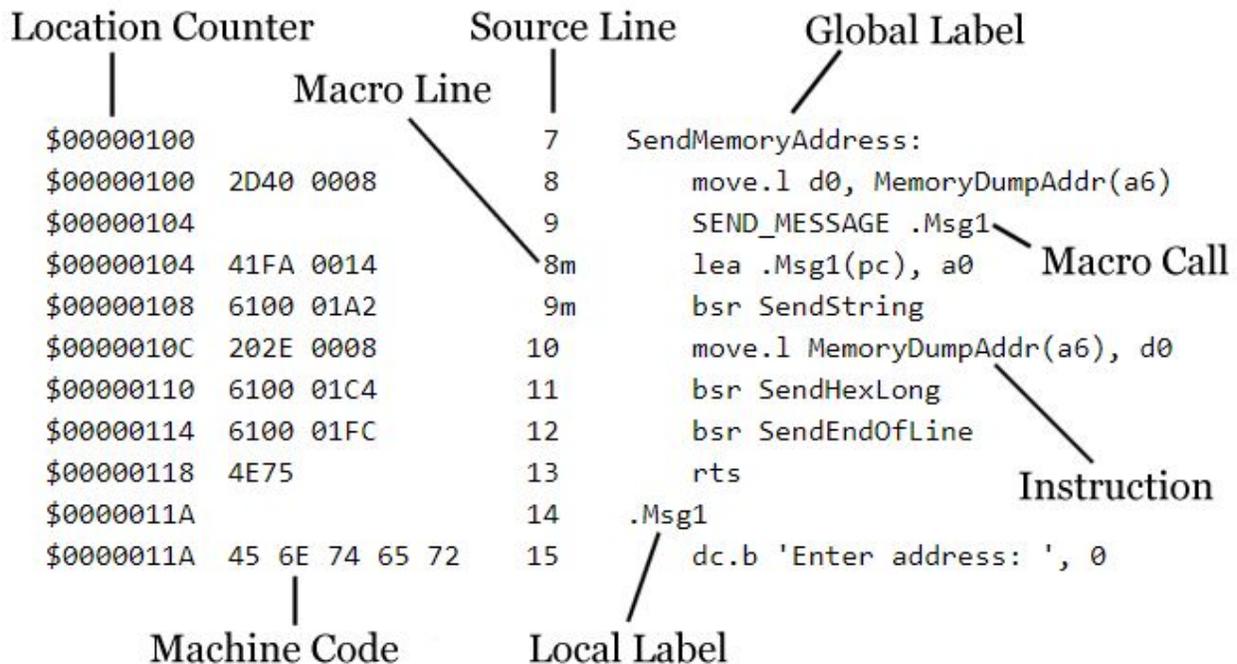
Listing files

LEA can produce a single listing for the entire run and/or individual listings for each file included in the run, depending on command line options. The default is to produce a whole-program listing in the same folder as the machine code output files. Currently, this is fixed as a subfolder called `Output` under the *project folder*, where the project folder is the folder containing the main source file specified on the command line. The whole-program listing is written to a file called `<project>.lis` where `<project>` is the name of the project specified on the command line (the default is 'Project').

⚠ LEA does not check if any output file exists before overwriting it. Therefore, any files in the `Output` subfolder might be deleted or overwritten when LEA is run.

Listing example and annotation

The following diagram shows the main features of the listing. Of particular importance is the *Location Counter* which shows the address in the target machine where the machine code is expected to be loaded.



Also note the Macro Line which has an `m` character to show a macro call. When a macro call is nested there will be one more `m` for each nested level. The maximum depth of macro nesting is determined by the number of `ms` that will fit in that space in the listing. This in turn is controlled by the *listing picture*. See the section entitled *Advanced Features* for more information.

Output files

LEA produces machine code in one or more formats. The default output formats are decided by the target CPU, but they can be overridden by using command line options. The following output formats are supported:

- Raw binary.
- Intel HEX format. See https://en.wikipedia.org/wiki/Intel_HEX
- Motorola SREC format. See <https://en.wikipedia.org/wiki/SREC>

Each of these formats can be output as a single contiguous file or split into high and low files. See the command line options for more details. These files are written to the same Output folder as the listing files and same overwrite warning applies to them.

Command line

LEA's command line interface takes a single filename (whose extension determines the target CPU) and zero or more options. The command line has the following format:

```
lea filename [/T][/L][/S][/R][/H][/C][/W][/I][/V][/D][/O][/F][/E][/Q][/Z]
```

The items enclosed in [] are optional arguments which specify the various assembler options. Most of these options have the general form */x=value* where *x* is one of the above options and *value* is either an integer, boolean or string. The */D* option is an exception; its value immediately follows it with no equals sign.



'-' can also be used in place of '/'.

Option	Type	Range	Default	Meaning
/T	int	1 to 20	4	Set listing tab stop. This controls the number of spaces used to represent a tab in the listing file.
/L	int	1 to 999	46	Set listing lines-per-page. This controls the number of lines on a page in the listing file before a page break / page title sequence appears.
/S	int	1 to 50	20	Set symbol dump field width. This controls the field width of symbols listed at the end of the listing file.
/R	int	1 to 999999	100000	Set maximum count for 'repeat' directive.
/H	int	1 to 128	32	Set Intel/Motorola Hex file record length.
/C	bool	true, false	false	Case sensitivity. When this option is enabled (true), LEA takes character case into account when looking up symbols and substituting macro arguments for parameters.
/W	bool	true, false	true	Generate whole program listing. If this option is enabled (true), LEA will generate a listing file for the whole program.
/I	bool	true, false	false	Generate individual listings. If this option is enabled (true), LEA will generate a listing file for every included source code file.
/F	bool	true, false	false	Display file, folder and project information. This option causes LEA to output verbose information regarding its use of files and folders. It can help when troubleshooting problems with files.
/E	string	CR, LF, CRLF	CRLF	Specify line-ending style for listing files.
/V	bool	true, false	true	Verbose output for errors and warnings.
/D	string			Predefine symbol(s). e.g., /DDEBUG=1, ADDRESS=\$1000
/P	string	-	Project	Set project name for listing file name and page header.

/O	string	B1, B2, I1, I2, M1, M2	Target CPU decides	Specify output file type(s). Any or all (or none) can be specified at the same time.
/Q	bool	true, false	true	Optimize move, add and sub to quick version if possible (68000 only).
/Z	bool	true, false	true	Optimize $\theta(A_n)$ to (A_n) (68000 only).

Source code layout

Although LEA can target more than one CPU, the format of any particular source code line is always the same:

```
[label[:]]      [order]      [comment]
```

A *label* is a symbol that takes on the value of the location counter at the start of the line it appears on. The label can end with an optional colon (:) which is solely for readability and is ignored by the assembler. Labels can be *global* or *local*. A global label starts with either A to Z or underscore (_) and can contain any number of characters A to Z, 0 to 9 or underscore. Global labels are used for defining *constants*, *variables* or *macros* and also for naming routines. Local labels begin with either a dot (.) or a colon (:) and are used as branch targets or local data labels within routines. Local labels have a *scope* which is defined by the surrounding global labels and do not exist outside that scope. The appearance of a global label begins a new scope for local labels. Note, though, that the local scope does *not* end when a global label is encountered at the start of a set, equ or macro directive.

An *order* is the generic name given to a *directive*, a *mnemonic*, or a *macro*.

A *directive* is an assembler command that instructs the assembler to perform some action. It usually does not generate any machine code, although some do (e.g., dc).

A *mnemonic* is a *keyword* used by a particular target CPU to refer to an instruction. For example, the 68000 keyword *move* is a mnemonic.

A *macro* is a recorded sequence of source code lines that can be invoked (expanded inline) by using the name of the macro as if it were a directive. For detailed information see the section entitled 'Macros'.

White space

The gaps between items on a line are made up of white space which is any number of tabs or spaces. Unlike some assemblers, LEA does not restrict white space usage. Any amount of whitespace can appear between items, including within expressions and operand lists. Some older assemblers would flag an error if, for example, there was a space in an operand list and would force the programmer to write difficult-to-read code with lots of items crammed together. In LEA, the only white space requirement is that spaces cannot appear inside names or numbers and there must be at least one tab or space before the order field, whether or not a label appears at the start of the line.

Comments

An entire line is treated as a comment if it begins with an asterisk (*) or a semicolon (;). A comment can also appear at the end of a line of code, in which case it should start with a semicolon.

```
* This is a comment.  
; This is also a comment.  
moveq #9, d0 ; This is also a comment.
```

⚠ Much of the time, comments can appear at the end of a line without a semicolon, but there are certain situations where that will cause an error. Because the assembler allows free use of white space in expressions, spurious text after an expression may appear like the expression is continuing. To be on the safe side, always use a semicolon.

```
move.l 10, 20 This comment is OK.  
move.l 10, 20 - This comment is not OK.  
move.l 10, 20 * This comment is not OK.
```

Expressions

Anywhere in the source code where a number can be used, an entire *expression* can be used. Expressions instruct LEA to perform a calculation in order to compute a number, so that the programmer does not have to compute the number and type it in. Note that these computations are performed by the assembler, not the target CPU. Very often, an entire expression generates only part of a single target CPU instruction. Consider this code snippet:

```
RAM    equ $80000
OFFSET equ $40
...
lea RAM+OFFSET, a0
```

Here, the `lea` instruction uses the expression `RAM+OFFSET` to make the code easier to read and modify.

Numeric values

All internal arithmetic is done using 64-bit integers. To define a numeric value in source code, simply type it in; the default is decimal. Any value up to the maximum for signed 64-bit arithmetic can be used, but the results must fall in the range suitable for the target CPU instructions they appear in.

Radixes

LEA supports the usual radixes. As mentioned above, decimal is the default. The following other radixes are possible by prefixing numbers with the relevant specifier:

Specifier	Radix	Digit range	Example
%	Binary	0..1	%10101001
@	Octal	0..7	@604705
\$	Hexadecimal	0..15	\$1234ABCD
'	Character	See ASCII constants	'Name'

Operators

LEA allows expressions to be constructed by using a variety of operators which are executed according to their precedence. For example, as is common in programming languages, the + operator has a lower precedence than *. This makes the * operate before the +. So:

```
X set 5 * 4 + 3 * 2
```

will be executed as if it had been written as:

```
X set (5 * 4) + (3 * 2) ; 20 + 6 = 26
```

Parentheses can be used to force the order of execution, for example:

```
X set 5 * (4 + 3) * 2 ; 5 * 7 * 2 = 70
```

The following is a list of supported operators and their meanings

+	Add
-	Subtract
*	Multiply
/	Divide
% (or MOD)	Modulus
&	Bitwise AND
AND	Logical AND
	Bitwise OR
OR	Logical OR
^	Bitwise Exclusive OR
EOR	Logical Exclusive OR
~	Bitwise NOT
NOT	Logical NOT
<<	Shift left
>>	Shift right
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
=	Equal to
<>	Not equal to
+	Unary plus
-	Unary minus
()	Parentheses

The following table shows operator precedence from lowest to highest:

Prec.	Operation	Symbols
1	Bitwise OR / Exclusive OR	^
	Logical OR / Exclusive OR	OR EOR
2	Bitwise AND	&
	Logical AND	AND
3	Relational	< <= > >= = <>
4	Add, subtract	+ -
5	Multiply, divide, modulus	* / % MOD
6	Shift left, shift right	<< >>
7	Unary plus, Unary minus, Bitwise NOT	+ - ~
	Logical NOT	NOT
8	Parentheses	()

Bitwise vs Logical operators

The operators for **and**, **or**, **exclusive-or** and **not** come in two flavours; *bitwise* and *logical*. These operate according to the following rules:

- The bitwise operators perform an operation on the individual bits of the values in the same way as the 68000's logical operations do.
- The logical operators perform boolean operations on the values by considering them as simply *true* or *false*.
- A value is considered to be true when it is non-zero and false when it is zero.
- True is represented by -1 and false by 0.

The following table shows how these operations compare to the C/C++ languages. Bear in mind that in C/C++, true is represented by 1, but LEA represents true by using -1.

Operation	LEA	C/C++
Bitwise and	&	&
Bitwise or		
Bitwise eor	^	^
Bitwise not	~	~
Logical and	AND	&&
Logical or	OR	
Logical eor	EOR	n/a
Logical not	NOT	!

Character strings

LEA supports two kinds of character strings depending on the context.

ASCII constants

In expressions, a numeric value can start with a single quote (') which indicates that the value is an *ASCII constant*. The characters in the string are encoded into the value as ASCII. For example:

```
'A' ; Value is 65 (hex $41)
'AB' ; Value is 16,706 (hex $4142)
'ABC' ; Value is 4,276,803 (hex $414243)
```

Since LEA internally uses 64-bit numbers, the longest ASCII constant can be eight characters long, but this is not compatible with any of the supplied target CPUs. For example, the 68000 only allows values up to 32 bits wide which would allow a four-character ASCII constant such as 'ABCD'. An example use of this feature is:

```
bsr CheckKeyboard
cmp.b #'b', d0 ; Was B pressed?
beq .KeyB
```

 Single quotes can appear in ASCII constants by 'doubling them up'. For example, the construct `' '' '` specifies a single `'` character.

String values

Some kinds of expressions also allow strings which consist of characters surrounded by single quotes. For example, the `include` and `incbin` directives expect a file name parameter. Strings are also allowed in some contexts during expression evaluation. They can be added with the `+` operator or compared with the relational operators. The result of string addition is another string which is the concatenation of the two strings. The result of string comparison is the value 0 (meaning false) or -1 (meaning true). An example follows:

```
HILO Macro String, Value
      if "\String" = "Hi"
x     set \Value >> 8
      else
x     set \Value & 255
      endif
      endm
```

 Some older assemblers accomplish this by using the special *if* statements `IFC` and `IFNC`. LEA does not currently support this syntax.

 Double quotes can appear in strings by 'doubling them up'. For example, the construct `"" ""` specifies a single `"` character.

System symbols

LEA defines the following symbols at the start of the run. These symbols can be used in expressions anywhere a regular symbol can be used. These are read-only assembler-controlled symbols that appear as constants in the target program.

<code>_YEAR</code>	The current year (e.g., 2019)
<code>_MONTH</code>	The current month (1 to 12)
<code>_DAY</code>	The current day (1 to 31)
<code>_HOUR</code>	The current hour (0 to 23)
<code>_MINUTE</code>	The current minute (0 to 59)
<code>_SECOND</code>	The current second (0 to 59)
<code>_WEEKDAY</code>	The current day of the week (0=Sunday to 6 = Saturday)
<code>_YEARDAY</code>	The current day of the year (1 to 366)
<code>_TRUE</code>	<code>0xFFFFFFFFFFFFFFFF</code> (-1)
<code>_FALSE</code>	<code>0x0000000000000000</code> (0)
<code>_REPCNT</code>	The value of the outermost repeat block loop counter. If no repeat block is active, this symbol is undefined.
<code>_LC</code>	The value of the location counter as it was at the start of the line. You can also use <code>*</code> to get the location counter.
<code>_RS</code>	The current value of the RS offset (68000 only). For more information see the section on the <code>rs</code> directive.

Macros

A macro is a recorded sequence of source code lines. They are first defined by using the `macro` directive and later expanded as many times as needed by invoking the macro as an order. Macros can have any number of parameters which can be filled from arguments specified after the macro at expansion time. Each time a macro is expanded, each line of the macro body is scanned for parameters which are replaced with the corresponding arguments. Furthermore, the special parameter `\@` is replaced by the *macro call counter* which is a counter that starts at zero and is incremented each time the macro is invoked.

This simple example demonstrates how to use named parameters in a macro:

```
MOVEIT    macro value, register
          moveq #\value, d\register
          endm
```

In this example, the named values are *value* and *register*. The macro would be invoked as follows with the number of arguments matching the number of parameters:

```
MOVEIT 1, 0
MOVEIT 5, 6
MOVEIT 7, 2
```

In the body of the macro, the parameters start with the `\` character and are substituted verbatim for the corresponding arguments specified when the macro is invoked. Thus, the above would generate the following code:

```
moveq #1, d0
moveq #5, d6
moveq #7, d2
```

An alternative is to use *automatic parameters* which are simply named `\1`, `\2`, `\3` etc. In this case, the example macro would be written as follows:

```
MOVEIT    macro
          moveq #\1, \2
          endm
```

This method is less readable but was provided for compatibility with some older assemblers.

Macro call counter

Following is an example of using the macro call counter mentioned above. In this case is it used to make a local label unique on every call of the macro:

```
MYLOOP    macro count
           moveq #\count, d7
           .\@    bsr SomeRoutine
                dbra d7, .\@
           endm
```

```
MYLOOP 9 ; Label will be .0
MYLOOP 5 ; Label will be .1
MYLOOP 7 ; Label will be .2
```

Macro parameters are substituted inside strings and comments as well. For example, the following:

```
FRUIT     macro
           dc.b '\@' ; \@
           endm

FRUIT
FRUIT
FRUIT
```

generates:

```
dc.b '0' ; 0
dc.b '1' ; 1
dc.b '2' ; 2
```

Directives

Directives instruct the assembler to perform certain actions. There are two kinds of directives in LEA, *built-in* and *custom*. Built-in directives are, as the name implies, built into the assembler. These are directives that tend to be useful regardless of the target CPU. Custom directives, on the other hand, are only useful when assembling for a particular CPU. They are provided by the target CPU implementation and only work when that CPU is selected as the target.

Summary of built-in directives

<code>include</code>	Includes another source file.
<code>incbin</code>	Includes a file as binary.
<code>equ</code>	Creates a constant symbol.
<code>set</code>	Creates and/or sets the value of a variable symbol.
<code>macro</code>	Begins a macro definition.
<code>endm</code>	Ends a macro definition.
<code>mexit</code>	Exits a macro prematurely.
<code>end</code>	Marks the end of the source code and stops assembly.
<code>if</code>	Begins an 'if' block.
<code>else</code>	Begins the 'else' clause in an if-block.
<code>endif</code>	Ends an 'if' block.
<code>while</code>	Begins a 'while' block.
<code>endw</code>	Ends a 'while' block.
<code>repeat</code> <code>rept</code>	Begins a 'repeat' block.
<code>endr</code>	Ends a 'repeat' block.
<code>tabstop</code>	Sets the tab stop in the listing file(s).
<code>list</code>	Turns the listing on.
<code>nolist</code>	Turns the listing off.

<code>expand</code>	Turns macro expansion listing on.
<code>noexpand</code>	Turns macro expansion listing off.
<code>org</code>	Sets the value of the location counter.
<code>section</code>	Switches to another section.
<code>print</code>	Prints a simple message to the output window and listing file(s).
<code>printf</code>	Prints a formatted message to the output window and listing file(s).
<code>fail</code>	Flags an error and displays an error message.

Detailed explanation of built-in directives

INCLUDE

[<label>] include <filename>

Includes another source file. The filename parameter is a string and as such must be enclosed in double quotes if it contains any spaces. Includes can be nested meaning that an included file is free to include other files. A file cannot include itself directly or indirectly. If a file is included more than once, assembly will abort with a fatal error. Example:

```
include "SourceFile.68K"
```

INCBIN

[<label>] incbin <filename>

Includes a file as binary. The filename parameter is a string and as such must be enclosed in double quotes if it contains any spaces. This directive copies the specified file directly into the output file(s) as raw bytes of data. If the file is larger than 128MB, assembly will abort with a fatal error. Even if the file is not too big, this directive has the potential to overrun the location counter beyond the maximum allowed for the target CPU. If this happens, assembly will abort with a fatal error. Example:

```
MyData: incbin "DataFile.dat"
```

EQU

<label> equ <expression>

Creates a constant label. The result of the expression is assigned to a new label which can subsequently be used in future expressions (or in previous expressions if forward references are allowed there). Once assigned, the value of this label is fixed and cannot be changed. This directive is useful for assigning meaningful symbolic names to values such as memory or hardware addresses. Example:

```
MemoryBase equ $40000
```

SET

<label> set <expression>

Creates and/or sets the value of a variable. The result of the expression is assigned to an assembler variable which can subsequently be used in future expressions (or in previous expressions if forward references are allowed there). Once assigned, the value of the variable is free to change later. This directive is useful for creating temporary variables such as counters which can be used in macros and conditional expressions. Example:

```
Count set 10  
Count set Count + 1
```

MACRO

<label> macro <argument> [, <argument>...]

Begins a macro definition which can later be used as if it were a directive in order to automatically create predefined blocks of code. Macros must end with the `endm` directive and their definitions cannot be nested (although a macro *can* invoke another macro). When the assembler encounters the `macro` directive, it stores (verbatim) everything between `macro` and `endm` in a macro list. The label at the start of the line specifies the name of the macro which will now work as a 'custom order'. This makes it work like a directive that has optional arguments.

For more information, see the section above entitled 'Macros'. Example:

```
Write macro addr, data, reg  
    lea \addr, a\reg
```

```
move #\data, (a\reg)
endm
```

ENDM

endm

Ends the definition of a macro which was started with the macro directive. No label is allowed to appear before `endm`.

MEXIT

mexit

Exits a macro prematurely as if `endm` had been encountered. This is useful when used with a conditional directive such as `if` to make the macro exit if a certain condition is met. It is similar to using `return` in a C/C++ function (e.g., after detecting an error condition). Example:

```
Store macro reg
if \reg > 7
    print "Register error"
    mexit
    clr.l (a\reg)
endif
endm
```

END

[<label>] end [<expression>]

When the `end` directive is encountered, assembly stops as if the end of the file had been reached. Any lines of text after the `end` directive are ignored. `end` can be followed by an optional expression which specifies the starting address for executing the program. This address is written to certain kinds of output files (e.g., SREC) to tell the target system what address to start executing the program at.

IF

[<label>] if <expression>

The `if` directive begins an 'if' block which allows conditional assembly of blocks of code. This feature is useful when different code needs to be created in different situations. For example, you might want to only produce certain debugging code when the 'debug' version of the program is being assembled. The expression after the `if` is evaluated and the block (up to the following `else` or `endif`) is only assembled if the result is *non-zero*. If an `else` appears in the block, the code between the `else` and the following `endif` is only assembled if the original expression evaluated to *zero*. Example:

```
if x = 10 OR y = 3
  print "true"
else
  print "false"
endif
```

If-blocks can be nested as follows:

```
if x = 10
  if y = 3
    print "x = 10, y = 3"
  else
    print "x = 10, y <> 3"
  endif
else
  print "x <> 10"
endif
```

⚠ All symbols used in `if` statements must already have been defined beforehand. Forward references are not allowed with this directive.

ELSE

[<label>] else

Begins the 'else' clause in an if-block. See `if` for more details.

ENDIF

[<label>] endif

Ends an 'if' block. See `if` for more details.

WHILE

[<label>] while <expression>

Begins a 'while' loop. The expression is evaluated and if the result is *non-zero*, the code up to the following `endw` is executed and the process is repeated at the while statement. Once the expression evaluates to *zero* the loop terminates. 'While' loops can be nested as long as there is one `endw` to match each `while`. Example:

```
count set 10
    while count > 0
        nop
count set count - 1
    endw
```

⚠ All symbols used in `while` statements must already have been defined beforehand. Forward references are not allowed with this directive.

⚠ There is currently no provision for exiting the while loop if the condition never evaluates to zero. In this case, LEA will continue to generate code until the location counter overruns. The command line on Windows allows Control-C to be used to abort the program if such a situation occurs.

ENDW

[<label>] endw

Ends a 'while' block. See `while` for more details.

REPEAT

[<label>] **repeat** <expression>

Begins a 'repeat' loop. The expression is evaluated and a variable called the *repeat counter* is created with a value of zero. The lines of code between the repeat and the following *endr* are then assembled, the counter is incremented, and process is repeated at the repeat statement while the counter is not equal to the result of the expression, at which time the loop terminates. 'Repeat' loops can be nested as long as there is one *endr* to match each *repeat*.

The repeat count can be accessed by using the *repeat count substitution operator* which is a question-mark. LEA scans all lines of code for the occurrence of a question-mark and if found, the outermost repeat count is substituted in place of the question-mark. Strings and comments are not affected by this substitution. Example:

```
count set 10
    repeat count * 2
    moveq #?, d0
    bsr Function
    endr
```

 All symbols used in *repeat* statements must already have been defined beforehand. Forward references are not allowed with this directive.

 The number of loops is limited to 100,000 unless overridden with the command line option /R. The command line on Windows allows Control-C to be used to abort the program if so desired.

 The alias *rept* can also be used for this directive. This is to allow compatibility with other assemblers.

ENDR

[<label>] endr

Ends a 'repeat' block. See `repeat` for more details.

TABSTOP

[<label>] tabstop <expression>

Sets the tab stop in the listing file(s). The expression is evaluated and the result is set as the number of spaces that will be used to represent a tab in the listing file. This overrides the default of 4 which itself may have been overridden by using the command line option `/T`. Example:

```
tabstop 8
```

LIST

[<label>] list

Turns the listing on. If the listing is turned on or off inside an included file, the previous state will be restored at the end of the included file.

NOLIST

[<label>] nolist

Turns the listing off. If the listing is turned on or off inside an included file, the previous state will be restored at the end of the included file.

EXPAND

[<label>] expand

Turns macro expansion listing on. If the listing is turned on or off inside an included file, the previous state will be restored at the end of the included file.

NOEXPAND

[<label>] noexpand

Turns macro expansion listing off. If the listing is turned on or off inside an included

file, the previous state will be restored at the end of the included file.

ORG

[<label>] org <expression>

Sets the value of the location counter. The location counter keeps track of the address in the target system where instructions and data will be stored. Each section has its own location counter so using `org` in a particular section will only change the location counter associated with that section. See the `section` directive for more information about sections.

⚠ All symbols used in `org` statements must already have been defined beforehand. Forward references are not allowed with this directive.

SECTION

[<label>] section <expression>

Switches to another section. Sections make it possible to output code or data for different purposes in different parts of the target system's memory, but without having to separate out all the source code for those parts. They are identified by numbers which in turn can be given symbolic names by using `equ` or `set`. Each section has its own location counter, so only the currently active section responds to activity involving the location counter such as code generation or the `org` directive.

By way of example, let's say the target system has several banks of memory which are used for different purposes, say code or data. You can generate the code and data all in one place by assigning a section for the code and another for the data. Example:

```
code equ 1
data equ 2

section code
org 0
lea data1, a0
bsr Routine

section data
org $100000
```

```

Data1:      dc.b 1, 2, 3, 4, 5, 6, 7, 8

            section code
            lea data2, a0
            bsr Routine

            section data
Data2:      dc.b 1, 2, 3, 4, 5, 6, 7, 8

```

In the above example, all the code specified in section "code" will be output to contiguous memory locations at the section's location counter (starting at its origin of 0), advancing the location counter by the required amount. All the data bytes specified in section "data" will be output to contiguous memory locations at the section's location counter (starting at its origin of \$100000), advancing *that* location counter by the required amount. This feature is particularly useful when creating macros or loops that generate some code and then some data that is used by that code. Sections can also be used for creating different streams of code from the same macro even though the code will appear on the target machine at completely different memory locations.



If the `section` directive is not used, the default section (zero) is assumed.

PRINT

[<label>] `print` <expression> [, <expression>...]

Prints a simple message to the output window and listing file(s). The directive is followed by one or more expressions separated by commas. The result of each expression is then 'printed' to the output window and the listing file(s). This simple message system is useful when debugging macros and the like as it makes it possible to see the contents of symbols or macro parameters.



String expressions are also allowed so that values can be labelled in the 'printout'.

Example:

```

PR      macro x
        print "Fred ", 7 * \x, " Bill ", 9 * \@
        endm

PR 4

```

PRINTF

[<label>] **printf** <string> [<expression>, ...]

Prints a formatted string to the output window and listing file(s). The directive is followed by a formatting string and then zero or more expressions separated by commas. The format string supports the same formatting specifiers as the C runtime library *printf* function. A few of them will be listed here, but there are many more which can be found here: <http://www.cplusplus.com/reference/cstdio/printf/>

Format specifier	Function	Argument
%u	Unsigned value	Number
%d	Signed value	Number
%X	Hexadecimal value	Number
%s	String value	String (see warning below)

Example: `printf "Value of X: %06d", x`

Result: `Value of X: 000009`

⚠ This is an advanced directive which (currently) can crash LEA if used incorrectly, because it directly passes the parameters to the C runtime library with no error checking. In particular, the `%s` specifier *must* match a string and floating-point specifiers cannot be used, nor can specifiers that *write to* memory.

FAIL

[<label>] **fail** [<expression>]

Flags an error and displays an error message. Assembly continues but the output file cannot be used due to the error. The error message is the result of the supplied expression which can be either a string or a number. Example:

```
if x > 9
fail "Invalid value"
endif
```


The final line is the *listing file padding*. This is a prefix used when a file name is output to the listing file.

[:-----]

Expression substitution

LEA allows entire expressions to be substituted anywhere in the source code by surrounding them with braces (`{}`). This powerful feature allows the result of any expression to appear anywhere in the source text as if the value had been typed there. Consider the following code snippet:

```
repeat 8
moveq #?, d?
endr
```

This will generate eight `moveq` instructions which load values from 0 to 7 into registers `d0` to `d7`. But what if we want the registers in the reverse order, `d7` to `d0`? We need to replace the final `?` with `7-?`. This is not possible unless we can somehow 'paste' the expression into the code, similar to the way the C preprocessor does 'token pasting'. In LEA, this is called *expression substitution*, and it is achieved as mentioned above, by placing the expression inside braces. So, we change the code to this:

```
repeat 8
moveq #?, d{7-?}
endr
```

LEA scans all text lines for matching braces. When it finds the `{7-?}`, it extracts the expression `7-?`, evaluates it, and replaces the entire `{7-?}` string with the result *before* assembling the line. So the line appears to the assembler as if the programmer had typed the result of the expression into the actual source code. The result of the above is:

```
repeat 8
moveq #0, d7
moveq #1, d6
moveq #2, d5
moveq #3, d4
moveq #4, d3
moveq #5, d2
moveq #6, d1
moveq #7, d0
endr
```

This rather simple example should give you some idea of how expression substitution can make it easier to produce code automatically. When combined with loops and

macros, it becomes a very powerful tool.

⚠ All symbols used in expression substitution must already have been defined beforehand. Forward references are not allowed. If the expression fails to evaluate, it will simply be left in the code unevaluated, without the braces. No error message will be generated unless the resulting code contains errors (which it usually will).

68000 Directives

The following directives are 68000-specific and are provided by the 68000 target CPU implementation. This means they only work when that CPU is selected as the target.

⚠ A full description of the 68000 instruction set is beyond the scope of this guide. There are many excellent sources of this information online.

Summary of 68000 directives

<code>even</code>	Moves the location counter to the next word boundary, if necessary.
<code>dc</code>	'Define Constant'. Defines one or more constant values.
<code>dcb</code>	'Define Constant Block'. Defines a block of constants with the same value.
<code>ds</code>	'Define Space'. Defines space at the current location.
<code>rs</code>	'Reserve Space'. Moves the RS offset either forwards or backwards.
<code>rsset</code>	Sets the value of the RS offset.
<code>rsreset</code>	Resets the value of the RS offset to zero.

Detailed explanation of 68000 directives

EVEN

[<label>] even

If the location counter is current set to an odd value, this directive increments it. This aligns it on a word boundary. When LEA generates 68000 instructions, this alignment is performed automatically, but there are times when the programmer needs to force alignment. Once such time is when a label appears at an odd address but is not on the same line as an instruction. Consider the following:

```
org $1000
dc.b 9, 8, 7
Label:    ; This label is at address $1003.
nop      ; This instruction is at address $1004.
```

After the odd number of `dc .b` bytes, the location counter is odd. Although the following `nop` will be word-aligned, the label will not. Any branch to the label will then cause an address error. This problem can be solved by using the `even` directive as follows:

```
        org $1000
        dc.b 9, 8, 7
        even
Label:   ; This label is at address $1004.
        nop ; This instruction is at address $1004.
```

DC

[<label>] dc[.size] <expression> [, <expression>...]

This directive defines constant values in the code at the current location. It is useful for defining tables and character strings. The directive can be followed by an optional size specifier to define bytes (`.b`), words (`.w`) or longwords (`.l`). The default is words. Any number of expressions can follow, separated by commas. A few examples follow:

```
Message: dc.b 'Hello world!', 13, 10, 0
Table:   dc.w 6 * 7, -48, 'AB', $1234, NOT 0, %1001
Vectors: dc.l Table, Message, 0
```

⚠ This directive does not support ASCII constants. The `'AB'` above is interpreted as a string just as if it had been written `"AB"`, and the characters are output as the individual words `$0041` and `$0042`.

DCB

[<label>] dcb[.size] <expression>, <expression>

This directive defines a block of data items at the current location, all with the same value. The directive can be followed by an optional size specifier to define bytes (.b), words (.w) or longwords (.l). The default is words. Two expressions must follow, the first giving the number of items to generate and the second, the value. A few examples follow:

```
Block1:  dcb.b 80, $40 ; Generate 80 bytes of value $40.  
Block2:  dcb.w 80, $40 ; Generate 80 words of value $0040.  
Block3:  dcb.l 80, $40 ; Generate 80 longs of value $00000040.
```

DS

[<label>] ds[.size] <expression>

This directive defines space in the code at the current location. It should not be confused with `rs` (see later). Example:

```
Empty1:  ds.b 80 ; Define 20 bytes of space.  
Empty2:  ds.w 80 ; Define 20 words of space (40 bytes).  
Empty3:  ds.l 80 ; Define 20 longs of space (80 bytes).
```

RS

[<rs-label>] rs[.size] <expression>

This directive reserves space in a *structure* by manipulating a value called the RS offset which keeps track of the current position in the structure being defined. The directive can be followed by an optional size specifier to reserve bytes (.b), words (.w) or longwords (.l). The default is words. First the expression is evaluated and the result is multiplied by the size (1, 2, or 4). If the result is positive, the current value of the RS offset is assigned to the label and then the offset is advanced by the result amount. If the result is negative, however, the RS offset is moved (backwards) by the amount *before* being assigned to the label. This makes it convenient for defining negative-going structures that are used with the `link` instruction. A few examples follow:

```

Label1:
    rsset 8
.x  rs.b 1          ; Local label .x is set to 8.
.y  rs.b 1          ; Local label .y is set to 9.
    lea PosData, a0
    move.b .x(a0), d0
    move.b .y(a0), d1
    ...
Label2:
    rsreset
.x  rs.w -1         ; Local label .x is set to -2.
.y  rs.w -1         ; Local label .y is set to -4.
.L  rs.l -1         ; Local label .L is set to -8.

    link a6, #_RS   ; Allocate the above structure on the
                    ; stack and set a6 to point to it.

    clr .x(a6)
    clr .y(a6)
    move.l #200, .L(a6)
    ...
    unlk a6         ; Deallocate the structure.

```

The above examples use local labels to demonstrate how to reuse structure element names in different structures. Global labels can also be used, but name clashes will occur if the same names are used in different structures.

RSSET

[<rs-label>] rsset <expression>

This directive sets the value of the RS offset to the result of an expression. See the description of the `rs` directive for more information about the RS offset. The optional label is assigned in the same way as for `rs`. Example:

```
rsset 10
```

RSRESET

[<rs-label>] rsreset

This directive resets the value of the *RS offset* to zero. It is equivalent to using the `rsset`

directive with zero for the parameter. See the description of the `rs` directive for more information about the RS offset. Example:

```
rsreset
```

Appendix

Formal expression grammar (BNF form)

```
<double-quote> ::= '"'
<single-quote> ::= "'"
<location-counter> ::= "*"
<ascii-char> ::= any ASCII byte value from 32 to 127
<alpha> ::= "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" |
           "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" |
           "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" | "a" |
           "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" |
           "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" |
           "t" | "u" | "v" | "w" | "x" | "y" | "z"
<bin-digit> ::= "0" | "1"
<oct-digit> ::= <bin-digit> | "2" | "3" | "4" | "5" | "6" | "7"
<dec-digit> ::= <oct-digit> | "8" | "9"
<hex-digit> ::= <dec-digit> | "A" | "B" | "C" | "D" | "E" | "F" |
               "a" | "b" | "c" | "d" | "e" | "f"
<bin-value> ::= "%" <bin-digit>+
<oct-value> ::= "@" <oct-digit>+
<hex-value> ::= "$" <hex-digit>+
<dec-value> ::= <dec-digit>+
<char-value> ::= <single_quote> <ascii-char>* <single_quote>
<ident-first> ::= "_" | <alpha>
<ident> ::= <ident-first> [<ident-first> | <dec-digit>]*
<global-label> ::= <ident>
<local-label> ::= "." | ":" <ident>
<label> ::= <local-label> | <global-label>
<unsigned-number> ::= <bin-value> | <oct-value> | <dec-value> |
                    <hex-value> | <char-value>
<string> ::= <double-quote> <char> <double-quote>
<orop> ::= "|" | "^" | "OR" | "EOR"
<andop> ::= "&" | "AND"
<addop> ::= "+" | "-"
<mulop> ::= "*" | "/" | "%" | "MOD"
<relop> ::= "<" | "<=" | ">" | ">=" | "=" | "<>"
<shiftopt> ::= "<<" | ">>"
<unop> ::= "+" | "-" | "~" | "NOT"
<factor> ::= "(" <expression> ")" | <location-counter> |
            <unsigned-number> | <ident> | <string>
```

<signed-factor> ::= <unop> <signed-factor> | <factor>
<shift-expression> ::= <signed-factor> [<shiftopt> <signed-factor>]*
<term> ::= <shift-expression> [<mulop> <shift-expression>]*
<simple-expression> ::= <term> [<addop> <term>]*
<relation> ::= <simple-expression> [<relop> <simple-expression>]*
<logical-term> ::= <relation> [<andop> <relation>]*
<expression> ::= <logical-term> [<orop> <logical-term>]*