LW Tool Chain

William Astle

LWTools Contributors

LW Tool Chain
by William Astle

by LWTools Contributors

Table of Contents

List of Tables

Chapter 1. Introduction

The LW tool chain provides utilities for building binaries for MC6809 and HD6309 CPUs. The tool chain includes a cross-assembler and a cross-linker which support several styles of output.

1.1. History

For a long time, I have had an interest in creating an operating system for the Coco3. I finally started working on that project around the beginning of 2006. I had a number of assemblers I could choose from. Eventually, I settled on one and started tinkering. After a while, I realized that assembler was not going to be sufficient due to lack of macros and issues with forward references. Then I tried another which handled forward references correctly but still did not support macros. I looked around at other assemblers and they all lacked one feature or another that I really wanted for creating my operating system.

The solution seemed clear at that point. I am a fair programmer so I figured I could write an assembler that would do everything I wanted an assembler to do. Thus the LWASM probject was born. After more than two years of on and off work, version 1.0 of LWASM was released in October of 2008.

As the aforementioned operating system project progressed further, it became clear that while assembling the whole project through a single file was doable, it was not practical. When I found myself playing some fancy games with macros in a bid to simulate sections, I realized I needed a means of assembling source files separately and linking them later. This spawned a major development effort to add an object file support to LWASM. It also spawned the LWLINK project to provide a means to actually link the files.

Chapter 2. Output Formats

The LW tool chain supports multiple output formats. Each format has its advantages and disadvantages. Each format is described below.

2.1. Raw Binaries

A raw binary is simply a string of bytes. There are no headers or other niceties. Both LWLINK and LWASM support generating raw binaries. ORG directives in the source code only serve to set the addresses that will be used for symbols but otherwise have no direct impact on the resulting binary.

2.2. DECB Binaries

A DECB binary is compatible with the LOADM command in Disk Extended Color Basic on the CoCo. They are also compatible with CLOADM from Extended Color Basic. These binaries include the load address of the binary as well as encoding an execution address. These binaries may contain multiple loadable sections, each of which has its own load address.

Each binary starts with a preamble. Each preamble is five bytes long. The first byte is zero. The next two bytes specify the number of bytes to load and the last two bytes specify the address to load the bytes at. Then, a string of bytes follows. After this string of bytes, there may be another preamble or a postamble. A postamble is also five bytes in length. The first byte of the postamble is $FF, the next two are zero, and the last two are the execution address for the binary.

Both LWASM and LWLINK can output this format.

2.3. ASCII Hexadecimal

This human-readable ASCII hexadecimal format consists of CR+LF terminated lines of ASCII text. Each line has the following structure: a zero-padded four-digit ASCII hex address, a colon separator, and one or more zero-padded two-digit hex values separated by commas. ASCII Hexadecimal format favors paragraph-aligned addresses (i.e. a least significant address nybble value of zero). During output, the number of hex values on each line are adjusted to align the address of the next line on a paragraph boundary. The sequence of addresses in the ASCII Hexadecimal file directly follows that of the source file; multiple ORG directives in the source code may result in out-of-sequence addresses in the ASCII Hexadecimal output.

LWASM can output this format since version 4.10.

2.4. Motorola S-Record

This ASCII format consists of a series of CR+LF terminated "records" of ASCII text. Each record has the following structure: a start-of-record character "S", an ASCII record type digit (0-9), a two-digit ASCII hex byte count, a four-digit ASCII hex address, an optional sequence of two-digit ASCII hex data values, and a two-digit ASCII hex checksum. The LW tool chain issues only S0, S1, S5 and S9 record types. S1 records are limited to maximum of 16 data bytes in length, and paragraph alignment of addresses is favored. The address sequence of the S-Records directly follows that of the source file; multiple ORG directives in the source code may result in out-of-sequence addresses in the S-Record output.

Motorola S-Record format is a standard ASCII format accepted by most memory device programming equipment. It is particularly useful when the assembled code output is destined to reside within an EPROM or Flash memory device, for example.

LWASM can output this format since version 4.10.

2.5. Intel Hex

This ASCII format consists of a series of CR+LF terminated "records" of ASCII text. Each record has the following structure: a start-of-record character ":", a two-digit ASCII hex byte count, a four-digit ASCII hex address, a two- digit ASCII hex record type, an optional sequence of two-digit ASCII hex data values, and a two-digit ASCII hex checksum. The LW tool chain issues only 00, and 01 Intel Hex record types. Data records are limited to maximum of 16 data bytes in length, and paragraph alignment of addresses is favored. The address sequence of the Intel hex records directly follows that of the source file; multiple ORG directives in the source code may result in out-of-sequence addresses in the Intel Hex output.

Intel Hex format is the other standard ASCII format accepted by most memory device programming equipment, it and the Motorola S-Record format are used for similar purposes.

LWASM can output this format since version 4.10.

2.6. OS9 Modules

Since version 2.5, LWASM is able to generate OS9 modules. The syntax is basically the same as for other assemblers. A module starts with the MOD directive and ends with the EMOD directive. The OS9 directive is provided as a shortcut for writing system calls.

LWASM does NOT provide an OS9Defs file. You must provide your own. Also note that the common practice of using "ifp1" around the inclusion of the OS9Defs file is discouraged as it is pointless and can

lead to unintentional problems and phasing errors. Because LWASM reads each file exactly once, there is no benefit to restricting the inclusion to the first assembly pass.

As of version 4.5, LWASM also implements the standard data/code address streams for OS9 modules. That means that between MOD and EMOD, any RMB, RMD, RMQ, or equivalent directives will move the data address ahead and leave the code address unmodified. Outside of an actual module, both the code and data addresses are moved ahead equally. That last bit is critical to understand because it means any directives that follow an EMOD directive may have different results than other assemblers.

Additionally, within a module body, the ORG directive sets only the data address, not the code address. However, outside a module body, ORG sets both addresses.

Both code and data addresses are reset to 0 by the MOD directive.

As of version 4.5, LWLINK also supports creation of OS9 modules.

2.7. Object Files

LWASM supports generating a proprietary object file format which is described in Chapter 6. LWLINK is then used to link these object files into a final binary in any of LWLINK's supported binary formats.

Object files also support the concept of sections which are not valid for other output types. This allows related code from each object file linked to be collapsed together in the final binary.

Object files are very flexible in that they allow references that are not known at assembly time to be resolved at link time. However, because the addresses of such references are not known at assembly time, there is no way for the assembler to deduce that an eight bit addressing mode is possible. That means the assember will default to using sixteen bit addressing whenever an external or cross-section reference is used.

As of LWASM 2.4, it is possible to force direct page addressing for an external reference. Care must be taken to ensure the resulting addresses are really in the direct page since the linker does not know what the direct page is supposed to be and does not emit errors for byte overflows.

It is also possible to use external references in an eight bit immediate mode instruction. In this case, only the low order eight bits will be used. Again, no byte overflows will be flagged.

Chapter 3. LWASM

The LWTOOLS assembler is called LWASM. This chapter documents the various features of the assembler. It is not, however, a tutorial on 6x09 assembly language programming.

3.1. Command Line Options

The binary for LWASM is called "lwasm". Note that the binary is in lower case. lwasm takes the following command line arguments.

`--6309`
`-3`

This will cause the assembler to accept the additional instructions available on the 6309 processor. This is the default mode; this option is provided for completeness and to override preset command arguments.

`--6800compat`

This is equivalent to `--pragma=6800compat`.

This will enable recognition of 6800 compatibility instructions.

`--6809`
`-9`

This will cause the assembler to reject instructions that are only available on the 6309 processor.

`--decb`
`-b`

Select the DECB output format target. Equivalent to `--format=decb`.

While this is the default output format currently, it is not safe to rely on that fact. Future versions may have different defaults. It is also trivial to modify the source code to change the default. Thus, it is recommended to specify this option if you need DECB output.

`--format=type`
`-f type`

Select the output format. Valid values are `obj` for the object file target, `decb` for the DECB LOADM format, `os9` for creating OS9 modules, `raw` for a raw binary, `hex` for ASCII hexadecminal format, `srec` for Motorola S-Record format, and `ihex` for Intel Hex format.

`--list[=file]`
`-l[file]`

Cause LWASM to generate a listing. If `file` is specified, the listing will go to that file. Otherwise it will go to the standard output stream. By default, no listing is generated. Unless `--symbols` is specified, the list will not include the symbol table.

`--symbols`
`-s`

Causes LWASM to generate a list of symbols when generating a listing. It has no effect unless a listing is being generated.

`--obj`

Select the proprietary object file format as the output target.

`--output=FILE`
`-o FILE`

This option specifies the name of the output file. If not specified, the default is `a.out`.

`--pragma=pragma`
`-p pragma`

Specify assembler pragmas. Multiple pragmas are separated by commas. The pragmas accepted are the same as for the PRAGMA assembler directive described below.

`--raw`
`-r`

Select raw binary as the output target.

`--includedir=path`
`-I path`

Add `path` to the end of the include path.

`--define=SYM[=VAL]`
`-D SYM[=VAL]`

Pre-defines the symbol SYM as either the specified VAL. If VAL is omitted, the symbol is defined as 1. The symbol will be defined as though it were defined using the SET directive in the assembly source. That means it can be overridden by a SET directive within the source code. Attempting to redefine SYM using EQU will result in a multiply defined symbol error.

`--help`
`-?`

Present a help screen describing the command line options.

`--usage`

Provide a summary of the command line options.

`--version`
`-V`

Display the software version.

`--debug`
`-d`

Increase the debugging level. Only really useful to people hacking on the LWASM source code itself.

3.2. Dialects

LWASM supports all documented MC6809 instructions as defined by Motorola. By default, this does not include any MC6800 compatibility instructions. As of LWASM 4.11, those compatibility instructions can be enabled using the *`--6800compat`* option or the *`6800compat`* pragma. It also supports all known HD6309 instructions. While there is general agreement on the pneumonics for most of the 6309 instructions, there is some variance with the block transfer instructions. TFM for all four variations seems to have gained the most traction and, thus, this is the form that is recommended for LWASM. However, it also supports COPY, COPY-, IMP, EXP, TFRP, TFRM, TFRS, and TFRR. It further adds COPY+ as a synomym for COPY, IMPLODE for IMP, and EXPAND for EXP.

By default, LWASM accepts 6309 instructions. However, using the *`--6809`* parameter, you can cause it to throw errors on 6309 instructions instead.

The standard addressing mode specifiers are supported. These are the hash sign ("#") for immediate mode, the less than sign ("<") for forced eight bit modes, and the greater than sign (">") for forced sixteen bit modes.

Additionally, LWASM supports using the asterisk ("*") to indicate base page addressing. This should not be used in hand-written source code, however, because it is non-standard and may or may not be present in future versions of LWASM.

3.3. Source Format

LWASM accepts plain text files in a relatively free form. It can handle lines terminated with CR, LF, CRLF, or LFCR which means it should be able to assemble files on any platform on which it compiles.

Each line may start with a symbol. If a symbol is present, there must not be any whitespace preceding it. It is legal for a line to contain nothing but a symbol.

The op code is separated from the symbol by whitespace. If there is no symbol, there must be at least one white space character preceding it. If applicable, the operand follows separated by whitespace. Following the opcode and operand is an optional comment.

It is important to note that operands cannot contain any whitespace except in the case of delimited strings. This is because the first whitespace character will be interpreted as the separator between the operand column and the comment. This behaviour is required for approximate source compatibility with other 6x09 assemblers.

A comment can also be introduced with a * or a ;. The comment character is optional for end of statement comments. However, if a symbol is the only thing present on the line other than the comment, the comment character is mandatory to prevent the assembler from interpreting the comment as an opcode.

For compatibility with the output generated by some C preprocessors, LWASM will also ignore lines that begin with a #. This should not be used as a general comment character, however.

The opcode is not treated case sensitively. Neither are register names in the operand fields. Symbols, however, are case sensitive.

As of version 2.6, LWASM supports files with line numbers. If line numbers are present, the line must start with a digit. The line number itself must consist only of digits. The line number must then be followed by either the end of the line or exactly one white space character. After that white space character, the lines are interpreted exactly as above.

3.4. Symbols

Symbols have no length restriction. They may contain letters, numbers, dots, dollar signs, and underscores. They must start with a letter, dot, or underscore.

LWASM also supports the concept of a local symbol. A local symbol is one which contains either a "?" or a "@", which can appear anywhere in the symbol. The scope of a local symbol is determined by a number of factors. First, each included file gets its own local symbol scope. A blank line will also be considered a local scope barrier. Macros each have their own local symbol scope as well (which has a side effect that you cannot use a local symbol as an argument to a macro). There are other factors as well. In general, a local symbol is restricted to the block of code it is defined within.

By default, unless assembling to the os9 target, a "$" in the symbol will also make it local. This can be controlled by the "dollarlocal" and "nodollarlocal" pragmas. In the absence of a pragma to the contrary, for the os9 target, a "$" in the symbol will not make it considered local while for all other targets it will.

3.5. Numbers and Expressions

Numbers can be expressed in binary, octal, decimal, or hexadecimal. Binary numbers may be prefixed with a "%" symbol or suffixed with a "b" or "B". Octal numbers may be prefixed with "@" or suffixed with "Q", "q", "O", or "o". Hexadecimal numbers may be prefixed with "$", "0x" or "0X", or suffixed with "H". No prefix or suffix is required for decimal numbers but they can be prefixed with "&" if desired. Any constant which begins with a letter must be expressed with the correct prefix base identifier or be prefixed with a 0. Thus hexadecimal FF would have to be written either 0FFH or $FF. Numbers are not case sensitive.

A symbol may appear at any point where a number is acceptable. The special symbol "*" can be used to represent the starting address of the current source line within expressions.

The ASCII value of a character can be included by prefixing it with a single quote (’). The ASCII values of two characters can be included by prefixing the characters with a quote (").

LWASM supports the following basic binary operators: +, -, *, /, and %. These represent addition, subtraction, multiplication, division, and modulus. It also supports unary negation and unary 1’s complement (- and ^ respectively). It is also possible to use ~ for the unary 1’s complement operator. For completeness, a unary positive (+) is supported though it is a no-op. LWASM also supports using |, &, and ^ for bitwise or, bitwise and, and bitwise exclusive or respectively.

Operator precedence follows the usual rules. Multiplication, division, and modulus take precedence over addition and subtraction. Unary operators take precedence over binary operators. Bitwise operators are lower precdence than addition and subtraction. To force a specific order of evaluation, parentheses can be used in the usual manner.

As of LWASM 2.5, the operators && and || are recognized for boolean and and boolean or respectively. They will return either 0 or 1 (false or true). They have the lowest precedence of all the binary operators.

3.6. Assembler Directives

Various directives can be used to control the behaviour of the assembler or to include non-code/data in the resulting output. Those directives that are not described in detail in other sections of this document are described below.

3.6.1. Data Directives

FCB *expr[,...]*
.DB *expr[,...]*
.BYTE *expr[,...]*

Include one or more constant bytes (separated by commas) in the output.

FDB *expr[,...]*
.DW *expr[,...]*
.WORD *expr[,...]*

Include one or more words (separated by commas) in the output.

FQB *expr[,...]*
.QUAD *expr[,...]*
.4BYTE *expr[,...]*

Include one or more double words (separated by commas) in the output.

FCC *string*
.ASCII *string*
.STR *string*

Include a string of text in the output. The first character of the operand is the delimiter which must appear as the last character and cannot appear within the string. The string is included with no modifications>

FCN *string*
.ASCIZ *string*
.STRZ *string*

Include a NUL terminated string of text in the output. The first character of the operand is the delimiter which must appear as the last character and cannot appear within the string. A NUL byte is automatically appended to the string.

FCS *string*
.ASCIS *string*
.STRS *string*

Include a string of text in the output with bit 7 of the final byte set. The first character of the operand is the delimiter which must appear as the last character and cannot appear within the string.

ZMB *expr*

Include a number of NUL bytes in the output. The number must be fully resolvable during pass 1 of assembly so no forward or external references are permitted.

ZMD *expr*

Include a number of zero words in the output. The number must be fully resolvable during pass 1 of assembly so no forward or external references are permitted.

ZMQ *expr*

Include a number of zero double-words in the output. The number must be fully resolvable during pass 1 of assembly so no forward or external references are permitted.

RMB *expr*
.BLKB *expr*
.DS *expr*
.RS *expr*

Reserve a number of bytes in the output. The number must be fully resolvable during pass 1 of assembly so no forward or external references are permitted. The value of the bytes is undefined.

RMD *expr*

Reserve a number of words in the output. The number must be fully resolvable during pass 1 of assembly so no forward or external references are permitted. The value of the words is undefined.

RMQ *expr*

Reserve a number of double-words in the output. The number must be fully resolvable during pass 1 of assembly so no forward or external references are permitted. The value of the double-words is undefined.

INCLUDEBIN *filename*

Treat the contents of *filename* as a string of bytes to be included literally at the current assembly point. This has the same effect as converting the file contents to a series of FCB statements and including those at the current assembly point.

If *filename* beings with a /, the file name will be taken as absolute. Otherwise, the current directory will be searched followed by the search path in the order specified.

Please note that absolute path detection including drive letters will not function correctly on Windows platforms. Non-absolute inclusion will work, however.

FILL *size*,*byte*

Insert *size* bytes of *byte*.

3.6.2. Address Definition

The directives in this section all control the addresses of symbols or the assembly process itself.

ORG *expr*

Set the assembly address. The address must be fully resolvable on the first pass so no external or forward references are permitted. ORG is not permitted within sections when outputting to object files. For target formats that include address information (decb, hex, srec, and ihex), an ORG

directive will re-start the address sequence within the output. When using the raw target format, ORG is used only to determine the addresses of symbols.

sym EQU *expr*
sym = *expr*

Define the value of *sym* to be *expr*.

sym SET *expr*

Define the value of *sym* to be *expr*. Unlike EQU, SET permits symbols to be defined multiple times as long as SET is used for all instances. Use of the symbol before the first SET statement that sets its value is undefined.

SETDP *expr*

Inform the assembler that it can assume the DP register contains *expr*. This directive is only advice to the assembler to determine whether an address is in the direct page and has no effect on the contents of the DP register. The value must be fully resolved during the first assembly pass because it affects the sizes of subsequent instructions.

This directive has no effect in the object file target.

ALIGN *expr*[,*value*]

Force the current assembly address to be a multiple of *expr*. If *value* is not specified, a series of NUL bytes is output to force the alignment, if required. Otherwise, the low order 8 bits of *value* will be used as the fill. The alignment value must be fully resolved on the first pass because it affects the addresses of subsquent instructions. However, *value* may include forward references; as long as it resolves to a constant for the second pass, the value will be accepted.

Unless *value* is specified as something like $12, this directive is not suitable for inclusion in the middle of actual code. The default padding value is $00 which is intended to be used within data blocks.

3.6.3. Conditional Assembly

Portions of the source code can be excluded or included based on conditions known at assembly time. Conditionals can be nested arbitrarily deeply. The directives associated with conditional assembly are described in this section.

All conditionals must be fully bracketed. That is, every conditional statement must eventually be followed by an ENDC at the same level of nesting.

Conditional expressions are only evaluated on the first assembly pass. It is not possible to game the assembly process by having a conditional change its value between assembly passes. Due to the underlying architecture of LWASM, there is no possible utility to IFP1 and IFP2, nor can they, as of LWASM 3.0, actually be implemented meaningfully. Thus there is not and never will be any equivalent of IFP1 or IFP2 as provided by other assemblers. Use of those opcodes will throw a warning and be ignored.

It is important to note that if a conditional does not resolve to a constant during the first parsing pass, an error will be thrown. This is unavoidable because the assembler must make a decision about which source to include and which source to exclude at this stage. Thus, expressions that work normally elsewhere will not work for conditions.

IFEQ *expr*

If *expr* evaluates to zero, the conditional will be considered true.

IFNE *expr*
IF *expr*

If *expr* evaluates to a non-zero value, the conditional will be considered true.

IFGT *expr*

If *expr* evaluates to a value greater than zero, the conditional will be considered true.

IFGE *expr*

If *expr* evaluates to a value greater than or equal to zero, the conditional will be considered true.

IFLT *expr*

If *expr* evaluates to a value less than zero, the conditional will be considered true.

IFLE *expr*

If *expr* evaluates to a value less than or equal to zero , the conditional will be considered true.

IFDEF *sym*

If *sym* is defined at this point in the assembly process, the conditional will be considered true.

IFNDEF *sym*

If *sym* is not defined at this point in the assembly process, the conditional will be considered true.

ELSE

If the preceding conditional at the same level of nesting was false, the statements following will be assembled. If the preceding conditional at the same level was true, the statements following will not be assembled. Note that the preceding conditional might have been another ELSE statement although this behaviour is not guaranteed to be supported in future versions of LWASM.

ENDC

This directive marks the end of a conditional construct. Every conditional construct must end with an ENDC directive.

3.6.4. OS9 Target Directives

This section includes directives that apply solely to the OS9 target.

OS9 *syscall*

This directive generates a call to the specified system call. *syscall* may be an arbitrary expression.

MOD *size*,*name*,*type*,*flags*,*execoff*,*datasize*

This tells LWASM that the beginning of the actual module is here. It will generate a module header based on the parameters specified. It will also begin calcuating the module CRC.

The precise meaning of the various parameters is beyond the scope of this document since it is not a tutorial on OS9 module programming.

EMOD

This marks the end of a module and causes LWASM to emit the calculated CRC for the module.

3.6.5. Miscelaneous Directives

This section includes directives that do not fit into the other categories.

INCLUDE *filename*
USE *filename*

Include the contents of *filename* at this point in the assembly as though it were a part of the file currently being processed. Note that if whitespace appears in the name of the file, you must enclose *filename* in quotes.

Note that the USE variation is provided only for compatibility with other assemblers. It is recommended to use the INCLUDE variation.

If *filename* begins with a "/", it is interpreted as an absolute path. If it does not, the search path will be used to find the file. First, the directory containing the file that contains this directive. (Includes within an included file are relative to the included file, not the file that included it.) If the file is not found there, the include path is searched. If it is still not found, an error will be thrown. Note that the current directory as understood by your shell or operating system is not searched.

END *[expr]*

This directive causes the assembler to stop assembling immediately as though it ran out of input. For the DECB target only, *expr* can be used to set the execution address of the resulting binary. For all other targets, specifying *expr* will cause an error.

ERROR *string*

Causes a custom error message to be printed at this line. This will cause assembly to fail. This directive is most useful inside conditional constructs to cause assembly to fail if some condition that is known bad happens. Everything from the directive to the end of the line is considered the error message.

WARNING *string*

Causes a custom warning message to be printed at this line. This will not cause assembly to fail. This directive is most useful inside conditional constructs or include files to alert the programmer to a deprecated feature being used or some other condition that may cause trouble later, but which may, in fact, not cause any trouble.

.MODULE *string*

This directive is ignored for most output targets. If the output target supports encoding a module name into it, *string* will be used as the module name.

As of version 3.0, no supported output targets support this directive.

3.7. Macros

LWASM is a macro assembler. A macro is simply a name that stands in for a series of instructions. Once a macro is defined, it is used like any other assembler directive. Defining a macro can be considered equivalent to adding additional assembler directives.

Macros may accept parameters. These parameters are referenced within a macro by the a backslash ("\") followed by a digit 1 through 9 for the first through ninth parameters. They may also be referenced by enclosing the decimal parameter number in braces ("{num}"). The special expansion "*" translates to the exact parameter string, including all parameters, passed to the macro. These parameter references are replaced with the verbatim text of the parameter passed to the macro. A reference to a non-existent parameter will be replaced by an empty string. Macro parameters are expanded everywhere on each source line. That means the parameter to a macro could be used as a symbol or it could even appear in a comment or could cause an entire source line to be commented out when the macro is expanded.

Parameters passed to a macro are separated by commas and the parameter list is terminated by any whitespace. This means that neither a comma nor whitespace may be included in a macro parameter.

Macro expansion is done recursively. That is, within a macro, macros are expanded. This can lead to infinite loops in macro expansion. If the assembler hangs for a long time while assembling a file that uses macros, this may be the reason.

Each macro expansion receives its own local symbol context which is not inherited by any macros called by it nor is it inherited from the context the macro was instantiated in. That means it is possible to use local symbols within macros without having them collide with symbols in other macros or outside the macro itself. However, this also means that using a local symbol as a parameter to a macro, while legal, will not do what it would seem to do as it will result in looking up the local symbol in the macro's symbol context rather than the enclosing context where it came from, likely yielding either an undefined symbol error or bizarre assembly results.

Note that there is no way to define a macro as local to a symbol context. All macros are part of the global macro namespace. However, macros have a separate namespace from symbols so it is possible to have a symbol with the same name as a macro.

Macros are defined only during the first pass. Macro expansion also only occurs during the first pass. On the second pass, the macro definition is simply ignored. Macros must be defined before they are used.

The following directives are used when defining macros.

macroname MACRO [NOEXPAND]

> This directive is used to being the definition of a macro called *macroname*. If *macroname* already exists, it is considered an error. Attempting to define a macro within a macro is undefined. It may work and it may not so the behaviour should not be relied upon.
>
> If NOEXPAND is specified, the macro will not be expanded in a program listing. Instead, all bytes emitted by all instructions within the macro will appear to be emitted on the line where the macro is invoked, starting at the address of the line of the invokation. If the macro uses ORG or other directives that define symbols or change the assembly address, these things will also be hidden (except in the symbol table) and the output bytes will appear with incorrect address attribution. Thus, NOEXPAND should only be used for macros that do not mess with the assembly address or otherwise define symbols that should be visible.

ENDM

> This directive indicates the end of the macro currently being defined. It causes the assembler to resume interpreting source lines as normal.

3.8. Structures

Structures are used to group related data in a fixed structure. A structure consists a number of fields, defined in sequential order and which take up specified size. The assembler does not enforce any means of access within a structure; it assumes that whatever you are doing, you intended to do. There are two pseudo ops that are used for defining structures.

structname STRUCT

This directive is used to begin the definition of a structure with name *structname*. Subsequent statements all form part of the structure definition until the end of the structure is declared.

ENDSTRUCT
ENDS

This directive ends the definition of the structure. ENDSTRUCT is the preferred form. Prior to version 3.0 of LWASM, ENDS was used to end a section instead of a structure.

Within a structure definition, only reservation pseudo ops are permitted. Anything else will cause an assembly error.

Once a structure is defined, you can reserve an area of memory in the same structure by using the structure name as the opcode. Structures can also contain fields that are themselves structures. See the example below.

```
tstruct2  STRUCT
f1        rmb 1
f2        rmb 1
          ENDSTRUCT

tstruct   STRUCT
field1    rmb 2
field2    rmb 3
field3    tstruct2
          ENDSTRUCT

          ORG $2000
var1      tstruct
var2      tstruct2
```

Fields are referenced using a dot (.) as a separator. To refer to the generic offset within a structure, use the structure name to the left of the dot. If referring to a field within an actual variable, use the variable's symbol name to the left of the dot.

You can also refer to the actual size of a structure (or a variable declared as a structure) using the special symbol sizeof{structname} where structname will be the name of the structure or the name of the variable.

Essentially, structures are a shortcut for defining a vast number of symbols. When a structure is defined, the assembler creates symbols for the various fields in the form structname.fieldname as well as the appropriate sizeof{structname} symbol. When a variable is declared as a structure, the assembler does the same thing using the name of the variable. You will see these symbols in the symbol table when the assembler is instructed to provide a listing. For instance, the above listing will create the following symbols (symbol values in parentheses): tstruct2.f1 (0), tstruct2.f2 (1), sizeof{tstruct2} (2), tstruct.field1 (0), tstruct.field2 (2), tstruct.field3 (5), tstruct.field3.f1 (5), tstruct.field3.f2 (6), sizeof{tstruct.field3} (2), sizeof{tstruct} (7), var1 {$2000}, var1.field1 {$2000}, var1.field2 {$2002}, var1.field3 {$2005}, var1.field3.f1 {$2005}, var1.field3.f2 {$2006}, sizeof(var1.field3} (2), sizeof{var1} (7), var2 ($2007), var2.f1 ($2007), var2.f2 ($2008), sizeof{var2} (2).

3.9. Object Files and Sections

The object file target is very useful for large project because it allows multiple files to be assembled independently and then linked into the final binary at a later time. It allows only the small portion of the project that was modified to be re-assembled rather than requiring the entire set of source code to be available to the assembler in a single assembly process. This can be particularly important if there are a large number of macros, symbol definitions, or other metadata that uses resources at assembly time. By far the largest benefit, however, is keeping the source files small enough for a mere mortal to find things in them.

With multi-file projects, there needs to be a means of resolving references to symbols in other source files. These are known as external references. The addresses of these symbols cannot be known until the linker joins all the object files into a single binary. This means that the assembler must be able to output the object code without knowing the value of the symbol. This places some restrictions on the code generated by the assembler. For example, the assembler cannot generate direct page addressing for instructions that reference external symbols because the address of the symbol may not be in the direct page. Similarly, relative branches and PC relative addressing cannot be used in their eight bit forms. Everything that must be resolved by the linker must be assembled to use the largest address size possible to allow the linker to fill in the correct value at link time. Note that the same problem applies to absolute address references as well, even those in the same source file, because the address is not known until link time.

It is often desired in multi-file projects to have code of various types grouped together in the final binary generated by the linker as well. The same applies to data. In order for the linker to do that, the bits that are to be grouped must be tagged in some manner. This is where the concept of sections comes in. Each chunk of code or data is part of a section in the object file. Then, when the linker reads all the object files, it coalesces all sections of the same name into a single section and then considers it as a unit.

The existence of sections, however, raises a problem for symbols even within the same source file. Thus, the assembler must treat symbols from different sections within the same source file in the same manner as external symbols. That is, it must leave them for the linker to resolve at link time, with all the limitations that entails.

In the object file target mode, LWASM requires all source lines that cause bytes to be output to be inside a section. Any directives that do not cause any bytes to be output can appear outside of a section. This includes such things as EQU or RMB. Even ORG can appear outside a section. ORG, however, makes no sense within a section because it is the linker that determines the starting address of the section's code, not the assembler.

All symbols defined globally in the assembly process are local to the source file and cannot be exported. All symbols defined within a section are considered local to the source file unless otherwise explicitly exported. Symbols referenced from external source files must be declared external, either explicitly or by asking the assembler to assume that all undefined symbols are external.

It is often handy to define a number of memory addresses that will be used for data at run-time but which need not be included in the binary file. These memory addresses are not initialized until run-time, either by the program itself or by the program loader, depending on the operating environment. Such sections are often known as BSS sections. LWASM supports generating sections with a BSS attribute set which causes the section definition including symbols exported from that section and those symbols required to resolve references from the local file, but with no actual code in the object file. It is illegal for any source lines within a BSS flagged section to cause any bytes to be output.

The following directives apply to section handling.

SECTION *name[,flags]*
SECT *name[,flags]*
.AREA *name[,flags]*

> Instructs the assembler that the code following this directive is to be considered part of the section *name*. A section name may appear multiple times in which case it is as though all the code from all the instances of that section appeared adjacent within the source file. However, *flags* may only be specified on the first instance of the section.
>
> *flags* is a comma separated list of flags. If a flag is "bss", the section will be treated as a BSS section and no statements that generate output are permitted.
>
> If the flag is "constant", the same restrictions apply as for BSS sections. Additionally, all symbols defined in a constant section define absolute values and will not be adjusted by the linker at link time. Constant sections cannot define complex expressions for symbols; the value must be fully defined at assembly time. Additionally, multiple instances of a constant section do not coalesce into a single addressing unit; each instance starts again at offset 0.
>
> If the section name is "bss" or ".bss" in any combination of upper and lower case, the section is assumed to be a BSS section. In that case, the flag *!bss* can be used to override this assumption.
>
> If the section name is "_constants" or "_constant", in any combination of upper and lower case, the section is assumed to be a constant section. This assumption can be overridden with the "!constant"

flag.

If assembly is already happening within a section, the section is implicitly ended and the new section started. This is not considered an error although it is recommended that all sections be explicitly closed.

ENDSECTION
ENDSECT

This directive ends the current section. This puts assembly outside of any sections until the next SECTION directive. ENDSECTION is the preferred form. Prior to version 3.0 of LWASM, ENDS could also be used to end a section but as of version 3.0, it is now an alias for ENDSTRUCT instead.

sym EXTERN
sym EXTERNAL
sym IMPORT

This directive defines *sym* as an external symbol. This directive may occur at any point in the source code. EXTERN definitions are resolved on the first pass so an EXTERN definition anywhere in the source file is valid for the entire file. The use of this directive is optional when the assembler is instructed to assume that all undefined symbols are external. In fact, in that mode, if the symbol is referenced before the EXTERN directive, an error will occur.

sym EXPORT
sym .GLOBL
EXPORT *sym*
.GLOBL *sym*

This directive defines *sym* as an exported symbol. This directive may occur at any point in the source code, even before the definition of the exported symbol.

Note that *sym* may appear as the operand or as the statement's symbol. If there is a symbol on the statement, that will take precedence over any operand that is present.

sym EXTDEP

This directive forces an external dependency on *sym*, even if it is never referenced anywhere else in this file.

3.10. Assembler Modes and Pragmas

There are a number of options that affect the way assembly is performed. Some of these options can only be specified on the command line because they determine something absolute about the assembly process. These include such things as the output target. Other things may be switchable during the assembly process. These are known as pragmas and are, by definition, not portable between assemblers.

LWASM supports a number of pragmas that affect code generation or otherwise affect the behaviour of the assembler. These may be specified by way of a command line option or by assembler directives. The directives are as follows.

PRAGMA *pragma[,...]*

Specifies that the assembler should bring into force all *pragma*s specified. Any unrecognized pragma will cause an assembly error. The new pragmas will take effect immediately. This directive should be used when the program will assemble incorrectly if the pragma is ignored or not supported.

*PRAGMA *pragma[,...]*

This is identical to the PRAGMA directive except no error will occur with unrecognized or unsupported pragmas. This directive, by virtue of starting with a comment character, will also be ignored by assemblers that do not support this directive. Use this variation if the pragma is not required for correct functioning of the code.

*PRAGMAPUSH *pragma[,...]*

This directive saves the current state of the specified pragma(s) for later retrieval. See discussion below for more information.

This directive will not throw any errors for any reason.

*PRAGMAPOP *pragma[,...]*

This directive restores the previously saved state of the specified pragma(s). See discussion below for more information.

This directive will not throw any errors for any reason.

Each pragma supported has a positive version and a negative version. The positive version enables the pragma while the negative version disables it. The negatitve version is simply the positive version with "no" prefixed to it. For instance, "pragma" vs. "nopragma". When only one version is listed below, its opposite can be obtained by prepending "no" if it is not present or removing "no" from the beginning if it is present.

Pragmas are not case sensitive.

6800compat

When in force, this pragma enables recognition of various compatibility instructions useful when assembling 6800 code. These compatibility instructions are assembled into equivalent 6809 instructions. This mode also includes several analogous instructions which are not strictly 6800 instructions but allow the similar style to be applied to 6809 specific features.

Technically, a compliant 6809 assembler must recognize these instructions by default since Motorola advertised the 6809 as being source compatible with the 6800. However, most source code does not require this compatibility and LWASM itself did not support these instructions prior to version 4.11 so this mode is disabled by default.

index0tonone

When in force, this pragma enables an optimization affecting indexed addressing modes. When the offset expression in an indexed mode evaluates to zero but is not explicity written as 0, this will replace the operand with the equivalent no offset mode, thus creating slightly faster code. Because of the advantages of this optimization, it is enabled by default.

cescapes

This pragma will cause strings in the FCC, FCS, and FCN pseudo operations to have C-style escape sequences interpreted. The one departure from the official spec is that unrecognized escape sequences will return either the character immediately following the backslash or some undefined value. Do not rely on the behaviour of undefined escape sequences.

importundefexport

This pragma is only valid for targets that support external references. When in force, it will cause the EXPORT directive to act as IMPORT if the symbol to be exported is not defined. This is provided for compatibility with the output of gcc6809 and should not be used in hand written code. Because of the confusion this pragma can cause, it is disabled by default.

undefextern

This pragma is only valid for targets that support external references. When in force, if the assembler sees an undefined symbol on the second pass, it will automatically define it as an external symbol. This automatic definition will apply for the remainder of the assembly process, even if the pragma is subsequently turned off. Because this behaviour would be potentially surprising, this pragma defaults to off.

The primary use for this pragma is for projects that share a large number of symbols between source files. In such cases, it is impractical to enumerate all the external references in every source file. This allows the assembler and linker to do the heavy lifting while not preventing a particular source module from defining a local symbol of the same name as an external symbol if it does not need the external symbol. (This pragma will not cause an automatic external definition if there is already a locally defined symbol.)

This pragma will often be specified on the command line for large projects. However, depending on the specific dynamics of the project, it may be sufficient for one or two files to use this pragma internally.

export

This pragma causes all symbols to be added to the export list automatically. This is useful when a large number of symbols need to be exported but you do not wish to include an EXPORT directive

for all of them. This is often useful on the command line but might be useful even inline with the PRAGMA directive if a large number of symbols in a row are to be exported.

dollarlocal

When set, a "$" in a symbol makes it local. When not set, "$" does not cause a symbol to be local. It is set by default except when using the OS9 target.

dollarnotlocal

This is the same as the "dollarlocal" pragma except its sense is reversed. That is, "dollarlocal" and "nodollarnotlocal" are equivalent and "nodollarlocal" and "dollarnotlocal" are equivalent.

pcaspcr

Normally, LWASM makes a distinction between PC and PCR in program counter relative addressing. In particular, the use of PC means an absolute offset from PC while PCR causes the assembler to calculate the offset to the specified operand and use that as the offset from PC. By setting this pragma, you can have PC treated the same as PCR.

shadow

When this pragma is in effect, it becomes possible to define a macro that matches an internal operation code. Thus, it makes it possible to redefine either CPU instructions or pseudo operations. Because this feature is of dubious utility, it is disabled by default.

nolist

Lines where this pragma is in effect will not appear in the assembly listing. Also, any symbols defined under this pragma will not show up in the symbol list. This is most useful in include files to avoid spamming the assembly listing with dozens, hundreds, or thousands of irrelevant symbols.

autobranchlength

One of the perennial annoyances for 6809 programmers is that the mneumonics for the short and long branch instructions are different (bxx vs. lbxx), which is at odds with the rest of the instruction set. This pragma is a solution to those annoying byte overflow errors that short branch instructions tend to aquire.

When this pragma is in effect, which is not the default, whenever any relative branch instruction is used, its size will be automatically determined based on the actual distance to the destination. In other words, one can write code with long or short branches everywhere and the assembler will choose a size for the branch.

Also, while this pragma is in effect, the > and < symbols can be used to force the branch size, analogous to their use for other instructions with < forcing 8 bit offsets and > forcing 16 bit offets.

Because this pragma leads to source that is incompatible with other assemblers, it is strongly recommended that it be invoked using the PRAGMA directive within the source code rather than on the command line or via the *PRAGMA directive. This way, an error will be raised if someone tries to assemble the code under a different assembler.

nosymbolcase
symbolnocase

Any symbol defined while this pragma is in force will be treated as case insensitive, regardless whether the pragma is in force when the symbol is referenced.

It is important to note that this pragma will not work as expected in all cases when using the object file assembly target. It is intended for use only when the assembler will be producing the final binary.

condundefzero

This pragma will cause the assembler to change the way it handles symbols in conditional expressions. Ordinarily, any symbol that is not defined prior to the conditional will throw an undefined symbol error. With this pragma in effect, symbols that are not yet defined at the point the conditional is encountered will be treated as zero.

This is not the default because it encourages poor code design. One should use the "IFDEF" or "IFNDEF" conditionals to test for the presence of a symbol.

It is important to note that if a symbol is defined but it does not yet evaluate to a constant value at the point where the conditional appears, the assembler will still complain about a non constant condition.

As a convenience, each input file has a pragma state stack. This allows, through the use of *PRAGMAPUSH and *PRAGMAPOP, a file to change a pragma state and then restore it to the precise state it had previously. If, at the end of an input file, all pragma states have not been popped, they will be removed from the stack. Thus, it is critical to employ *PRAGMAPOP correctly. Because each input file has its own pragma stack, using *PRAGMAPUSH in one file and *PRAGMAPOP in another file will not work.

Pragma stacks are more useful in include files, in particular in conjunction with the nolist pragma. One can push the state of the nolist pragma, engage the nolist pragma, and then pop the state of the nolist pragma at the end of the include file. This will cause the entire include file to operate under the nolist pragma. However, if the file is included while nolist is already engaged, it will not undo that state.

Chapter 4. LWLINK

The LWTOOLS linker is called LWLINK. This chapter documents the various features of the linker.

4.1. Command Line Options

The binary for LWLINK is called "lwlink". Note that the binary is in lower case. lwlink takes the following command line arguments.

`--decb`
`-b`

Selects the DECB output format target. This is equivalent to `--format=decb`

`--output=FILE`
`-o FILE`

This option specifies the name of the output file. If not specified, the default is `a.out`.

`--format=TYPE`
`-f TYPE`

This option specifies the output format. Valid values are `decb` and `raw`

`--raw`
`-r`

This option specifies the raw output format. It is equivalent to `--format=raw` and `-f raw`

`--script=FILE`
`-s`

This option allows specifying a linking script to override the linker's built in defaults.

`--section-base=SECT=BASE`

Cause section SECT to load at base address BASE. This will be prepended to the built-in link script. It is ignored if a link script is provided.

`--map=FILE`
`-m FILE`

This will output a description of the link result to FILE.

`--library=LIBSPEC`
`-l LIBSPEC`

Load a library using the library search path. LIBSPEC will have "lib" prepended and ".a" appended.

`--library-path=DIR`
`-L DIR`

Add DIR to the library search path.

`--debug`
`-d`

This option increases the debugging level. It is only useful for LWTOOLS developers.

`--help`
`-?`

This provides a listing of command line options and a brief description of each.

`--usage`

This will display a usage summary of each command line option.

`--version`
`-V`

This will display the version of LWLINK.

4.2. Linker Operation

LWLINK takes one or more files in supported input formats and links them into a single binary. Currently supported formats are the LWTOOLS object file format and the archive format used by LWAR. While the precise method is slightly different, linking can be conceptualized as the following steps.

1. First, the linker loads a linking script. If no script is specified, it loads a built-in default script based on the output format selected. This script tells the linker how to lay out the various sections in the final binary.
2. Next, the linker reads all the input files into memory. At this time, it flags any format errors in those files. It constructs a table of symbols for each object at this time.
3. The linker then proceeds with organizing the sections loaded from each file according to the linking script. As it does so, it is able to assign addresses to each symbol defined in each object file. At this time, the linker may also collapse different instances of the same section name into a single section by appending the data from each subsequent instance of the section to the first instance of the section.
4. Next, the linker looks through every object file for every incomplete reference. It then attempts to fully resolve that reference. If it cannot do so, it throws an error. Once a reference is resolved, the value is placed into the binary code at the specified section. It should be noted that an incomplete reference can reference either a symbol internal to the object file or an external symbol which is in the export list of another object file.
5. If all of the above steps are successful, the linker opens the output file and actually constructs the binary.

4.3. Linking Scripts

A linker script is used to instruct the linker about how to assemble the various sections into a completed binary. It consists of a series of directives which are considered in the order they are encountered.

The sections will appear in the resulting binary in the order they are specified in the script file. If a referenced section is not found, the linker will behave as though the section did exist but had a zero size, no relocations, and no exports. A section should only be referenced once. Any subsequent references will have an undefined effect.

All numbers are in linking scripts are specified in hexadecimal. All directives are case sensitive although the hexadecimal numbers are not.

A section name can be specified as a "*", then any section not already matched by the script will be matched. The "*" can be followed by a comma and a flag to narrow the section down slightly, also. If the flag is "!bss", then any section that is not flagged as a bss section will be matched. If the flag is "bss", then any section that is flagged as bss will be matched.

The following directives are understood in a linker script.

sectopt *`section`* padafter *`byte,...`*

> This will cause the linker to append the specified list of byte values (specified in hexadecimal separated by commas) to the end of the named section. This is done once all instances of the specified section are collected together. This has no effect if the specified section does not appear anywhere in any of the objects specified for linking.
>
> If code depends on the presence of this padding somewhere, it is sufficient to include an empty section of the specified name in the object that depends on it.

define basesympat *`string`*

> This causes the linker to define a symbol for the ultimate base address of each section using the pattern specified by *`string`*. In the string, %s can appear exactly once and will be replaced with the section name. The base address is calculated after all instances of each section have been collapsed together.
>
> It should be noted that if none of the objects to be linked contains a particular section name, there will be no base symbol defined for it, even if it is listed explicitly in the link script. If code depends on the presence of these symbols, it is sufficient to include an empty section of the specified name in the object that depends on it.
>
> If the pattern resolves to the same string for multiple sections, the results are undefined.

define lensympat *string*

This causes the linker to define a symbol for the ultimate length of each section using the pattern specified by *string*. In the string, %s can appear exactly once and will be replaced with the section name. The length is calculated after all instances of a section have been collapsed together.

It should be noted that if none of the objects to be linked contains a particular section name, there will be no length symbol defined for it, even if it is listed explicitly in the link script. If code depends on the presence of these symbols, it is sufficient to include an empty section of the specified name in the object that depends on it.

If the pattern resolves to the same string for multiple sections, the results are undefined.

section *name* load *addr*

This causes the section *name* to load at *addr*. For the raw target, only one "load at" entry is allowed for non-bss sections and it must be the first one. For raw targets, it affects the addresses the linker assigns to symbols but has no other affect on the output. bss sections may all have separate load addresses but since they will not appear in the binary anyway, this is okay.

For the decb target, each "load" entry will cause a new "block" to be output to the binary which will contain the load address. It is legal for sections to overlap in this manner - the linker assumes the loader will sort everything out.

section *name* high *addr*

This causes the section *name* to load with its end address just below *addr*. Subsequent sections are loaded at progressively lower addresses. This may lead to inefficient file encoding for some targets. As of this writing, it will also almost certainly do the wrong thing for a raw target.

This is useful for aligning a block of code with high memory. As an example, if the total size of a section is $100 bytes and a high address of $FE00 is specified, the section will actually load at $FD00.

section *name*

This will cause the section *name* to load after the previously listed section.

entry *addr or sym*

This will cause the execution address (entry point) to be the address specified (in hex) or the specified symbol name. The symbol name must match a symbol that is exported by one of the object files being linked. This has no effect for targets that do not encode the entry point into the resulting file. If not specified, the entry point is assumed to be address 0 which is probably not what

you want. The default link scripts for targets that support this directive automatically starts at the beginning of the first section (usually "init" or "code") that is emitted in the binary.

pad *size*

This will cause the output file to be padded with NUL bytes to be exactly *size* bytes in length. This only makes sense for a raw target.

4.4. Format Specific Linking Notes

Some formats require special information to be able to generate actual binaries. If the specific format you are interested in is not listed in this section, then there is nothing special you need to know about to create a final binary.

4.4.1. OS9 Modules

OS9 modules need to embed several items into the module header. These items are the type of module, the langauge of the module, the module attributes, the module revision number, the data size (bss), and the execution offset. These are all either calculated or default to reasonable values.

The data size is calcuated as the sum of all sections named "bss" or ".bss" in all object files that are linked together.

The execution offset is calculated from the address of the special symbol "__start" which must be an exported (external) symbol in one of the objects to be linked.

The type defaults to "Prgrm" or "Program module". The language defaults to "Objct" or "6809 object code". Attributes default to enabling the re-entrant flag. And finally, the revision defaults to zero.

The embedded module name is the output filename. If the output filename includes more than just the filename, this will probably not be what you want.

The type, language, attributes, revision, and module name can all be overridden by providing a special section in exactly one of the object files to be linked. This section is called "__os9" (note the two underscores). To override the type, language, attributes, or revision values, define a non-exported symbol in this section called "type", "lang", "attr", or "rev" respectively. Any other symbols defined are ignored. To override the module name, include as the only actual code in the section a NUL terminated string (the FCN directive is useful for this). If there is no code in the section or it beings with a NUL, the default name will be used. Any of the preceeding that are not defined in the special section will retain their default values.

The built-in link script for OS9 modules will place the following sections, in order, in the module: "code", ".text", "data", ".data". It will merge all sections with the name "bss" or ".bss" into the "data" section. All other section names are ignored. What this means is that you must define your data variables in the a section called "bss" or ".bss" even though you will be refencing them all as offsets from U. This does have the unpleasant side effect that all BSS references will end up being 16 bit offsets because the assembler cannot know what the offset will be once the linker is finished its work. Thus, if the tightest possible code is required, having LWASM directly output the module is a better choice.

While the built-in link script is probably sufficient for most purposes, you can provide your own script. If you provide a custom link script, you must start your code and data sections at location 000D to accommodate the module header. Otherwise, you will have an incorrect location for the execution offset. You must use the ENTRY directive in the script to define the entry point for the module.

It should also be obvious from the above that you cannot mix the bss (rmb) definitions with the module code when linking separately. Those familiar with typical module creation will probably find this an unpleasant difference but it is unavoidable.

It should also be noted that direct page references should also be avoided because you cannot know ahead of time whether the linker is going to end up putting a particular variable in the first 256 bytes of the module's data space. If, however, you know for certain you will have less than 256 bytes of defined data space across all of the object files that will be linked, you can instead use forced DP addressing for your data addresses instead of the ,u notation. When linking with 3rd party libraries, this practice should be avoided. Also, when creating libraries, always use the offset from U technique.

Chapter 5. Libraries and LWAR

LWTOOLS also includes a tool for managing libraries. These are analogous to the static libraries created with the "ar" tool on POSIX systems. Each library file contains one or more object files. The linker will treat the object files within a library as though they had been specified individually on the command line except when resolving external references. External references are looked up first within the object files within the library and then, if not found, the usual lookup based on the order the files are specified on the command line occurs.

The tool for creating these libary files is called LWAR.

5.1. Command Line Options

The binary for LWAR is called "lwar". Note that the binary is in lower case. The options lwar understands are listed below. For archive manipulation options, the first non-option argument is the name of the archive. All other non-option arguments are the names of files to operate on.

`--add`
`-a`

This option specifies that an archive is going to have files added to it. If the archive does not already exist, it is created. New files are added to the end of the archive.

`--create`
`-c`

This option specifies that an archive is going to be created and have files added to it. If the archive already exists, it is truncated.

`--merge`
`-m`

If specified, any files specified to be added to an archive will be checked to see if they are archives themselves. If so, their constituent members are added to the archive. This is useful for avoiding archives containing archives.

`--list`
`-l`

This will display a list of the files contained in the archive.

`--debug`
`-d`

This option increases the debugging level. It is only useful for LWTOOLS developers.

`--help`
`-?`

This provides a listing of command line options and a brief description of each.

`--usage`

This will display a usage summary of each command line option.

`--version`
`-V`

This will display the version of LWLINK. of each.

Chapter 6. Object Files

LWTOOLS uses a proprietary object file format. It is proprietary in the sense that it is specific to LWTOOLS, not that it is a hidden format. It would be hard to keep it hidden in an open source tool chain anyway. This chapter documents the object file format.

An object file consists of a series of sections each of which contains a list of exported symbols, a list of incomplete references, and a list of "local" symbols which may be used in calculating incomplete references. Each section will obviously also contain the object code.

Exported symbols must be completely resolved to an address within the section it is exported from. That is, an exported symbol must be a constant rather than defined in terms of other symbols.

Each object file starts with a magic number and version number. The magic number is the string "LWOBJ16" for this 16 bit object file format. The only defined version number is currently 0. Thus, the first 8 bytes of the object file are 4C574F424A313600

Each section has the following items in order:

- section name
- flags
- list of local symbols (and addresses within the section)
- list of exported symbols (and addresses within the section)
- list of incomplete references along with the expressions to calculate them
- the actual object code (for non-BSS sections)

The section starts with the name of the section with a NUL termination followed by a series of flag bytes terminated by NUL. There are only two flag bytes defined. A NUL (0) indicates no more flags and a value of 1 indicates the section is a BSS section. For a BSS section, no actual code is included in the object file.

Either a NULL section name or end of file indicate the presence of no more sections.

Each entry in the exported and local symbols table consists of the symbol (NUL terminated) followed by two bytes which contain the value in big endian order. The end of a symbol table is indicated by a NULL symbol name.

Each entry in the incomplete references table consists of an expression followed by a 16 bit offset where the reference goes. Expressions are defined as a series of terms up to an "end of expression" term. Each

term consists of a single byte which identifies the type of term (see below) followed by any data required by the term. Then end of the list is flagged by a NULL expression (only an end of expression term).

Table 6-1. Object File Term Types

TERMTYPE	Meaning
00	end of expression
01	integer (16 bit in big endian order follows)
02	external symbol reference (NUL terminated symbol name follows)
03	local symbol reference (NUL terminated symbol name follows)
04	operator (1 byte operator number)
05	section base address reference
FF	This term will set flags for the expression. Each one of these terms will set a single flag. All of them should be specified first in an expression. If they are not, the behaviour is undefined. The byte following is the flag. Flag 01 indicates an 8 bit relocation. Flag 02 indicates a zero-width relocation (see the EXTDEP pseudo op in LWASM).

External references are resolved using other object files while local references are resolved using the local symbol table(s) from this file. This allows local symbols that are not exported to have the same names as exported symbols or external references.

Table 6-2. Object File Operator Numbers

Number	Operator
01	addition (+)
02	subtraction (-)
03	multiplication (*)
04	division (/)
05	modulus (%)
06	integer division (\) (same as division)
07	bitwise and
08	bitwise or
09	bitwise xor
0A	boolean and
0B	boolean or
0C	unary negation, 2's complement (-)
0D	unary 1's complement (^)

An expression is represented in a postfix manner with both operands for binary operators preceding the operator and the single operand for unary operators preceding the operator.