ASM11

A two-pass absolute macro cross-assembler for the 68HC11

Quick Reference Guide

Last Update: May 29, 2019 for ASM11 v9.83

Command-Line Syntax and Options

`ASM11 [-option [...]] [[@]filespec [...]] [>errfile]`

- *option*(s) may appear before, in between or after *filespec*(s).
- *option*(s) specified apply to all files assembled, regardless of command line placement.
- Text file(s) containing list(s) of files to be processed may be specified by naming the text file on the command line, prefixed with a «@» character. These text files may not contain command line options.
- *filespec*(s) may include wildcard characters (?,*). Wildcards are not allowed in *filespec*(s) that are prefixed with a «@» but are allowed in filespecs inside @files.
- If the file extension for a source *filespec* is omitted, the extension «.ASM» is assumed (see description of the `-R.ext` option below).
- Assembler errors may be redirected to *errfile* using standard DOS output redirection syntax. This capability may be used in conjunction with, or as an alternative to the **`-E+`** option.
- Any label can hold a value that is 32-bit long. Even though the CPU cannot understand numbers larger than 16-bit for data or addressing, the ability to have 32-bit labels allows keeping constants that are larger than 16-bit for use in later constant calculations. Decimal numbers are signed; the largest number is +/-2147483647. Hex or binary numbers are unsigned and can go up to the full 32-bit value (2^{32}-1). For example, a symbol holding the crystal frequency of operation can be expressed with Hz detail to be used later to derive other constant values (such as cycle-based delays). *The 32-bit capability is NOT available in the DOS version.*
- The assembler will set the DOS ERRORLEVEL variable when it terminates as indicated:
  - 0 No error, assembly of last file was successful
  - 1 System error (hardware I/O failure, out of disk space, etc.), or **`-W`** option failure
  - 2 Error(s) generated (or Escape pressed) during assembly of last file
  - 3 Warning(s) generated during assembly of last file
  - 4 Assembler was not started (help screen displayed, **`-W`** option used with success)
    1. No file(s) found

Option	Default	Description
`-C``[±]`	`-C-`	Label case sensitivity: + = case sensitive (See also `#CASEON`/`#CASEOFF`)
`-D``label [:expr]`		Use up to one hundred times to define symbols either for conditional assembly (e.g., `IFDEF`, `IFNDEF`, and `IF` directives) or normal use. Symbols are always uppercase (regardless of **`-C`** option). If they are not followed by a value (or expression) they assume the value zero. Expression is limited to 19 characters. Character constants should not contain spaces, and they are converted to uppercase. Cannot be saved with **`-W`**.

-E[±]	-E-	Generate *.ERR file (one for each file assembled). *.ERR files are not generated for file(s) that do not contain errors.
-EH[±]	-EH+	If **-E+** is in effect, hide (do not display) error messages on screen.
-EXP[±]	-EXP-	When on, an .EXP file is created containing all symbols defined with an EXP rather than an EQU pseudo-opcode. The resulting file can then be used as an #INCLUDE file for other programs. This allows for automatic creation of include files with exported global symbols.
-F:symb **-F:**num		During assembly, if it finds the given symbol, it prints a 'Hint' message showing the file and line number when that symbol defines or redefines its value. You can optionally use * anywhere inside the symbol to match all symbols that include the one characters given. This is very useful for debugging hard to locate errors of where exactly in the source code a symbol acquired its value. Alternatively, you can give it a number (either in decimal or hex format the way the assembler understands numbers). In this case, the 'Hint' message will show the file and line number when that memory address was occupied by either data or code. This is very useful to help you resolve overlap type errors. You may use either : or = with this option after the -F. This option cannot be saved with the -W switch
-F2[±]	-F2-	Forces a P&E 16-bit map when in MMU mode. Useful to overcome bugs in certain P&E products that do not handle MMU addresses correctly. This option cannot be saved with the -W switch
-FD[±]	-FD-	When on, the assembler uses a fake/fixed date (specifically, Jan 1, 2011) for the internal symbols :YEAR, :MONTH, and :DATE. It does not falsify the date shown on the listing header, however. This option is useful to let you always get the same S19 CRC value (shown both at the end of the listing file, and on the command-line next to each successfully assembled file), even if you use the :YEAR, :MONTH, and :DATE internal symbols in your source code, which based on the compilation date of your program would normally alter the resulting S19 CRC. This would, in turn, make it more difficult to quickly check if your program produces the same, or a different binary, since last time you checked. Keeping a record in your source of the most recent S19 CRC produced with the -FD option, let's you know if something has (perhaps, inadvertently) changed. Without the -FD option, you can't be sure if it's just the date that changed, or something else. *WARNING: Do NOT include this option in batch or makefiles that compile your programs automatically, or you risk producing consistently misdated firmware.*

		It should only be used for manual verification purposes. It's not by accident this option cannot be saved with the -W switch.
-FE[±]	-FE-	Converts warnings to error messages. This option cannot be saved with the -W switch
-FH[±]	-FH-	Forces hidden macros in listings (#HIDEMACROS) and ignores all #SHOWMACROS directives. This option cannot be saved with the -W switch
-FI[±]	-FI-	Forces display of included filenames as hints. This option cannot be saved with the -W switch
-FL[±]	-FL-	Ignores all #LISTOFF or #NOLIST directives. This option cannot be saved with the -W switch
-FM[±]	-FM-	Ignores all #MAPOFF directives. This option cannot be saved with the -W switch
-FQ[±]	-FQ-	Does not show the assembly progress. Slightly better speed. For use with IDEs and makefiles. This option cannot be saved with the -W switch
-FW[±]	-FW-	Converts warnings to harmless messages. No error code is returned and warnings are not counted. This option cannot be saved with the -W switch
-FX[±]	-FX-	Enable macro line number display in FATAL, ERROR, WARNING, MESSAGE, and HINT directives. The default is off for less cluttered display. This option cannot be saved with the -W switch
-Ix		Define default INCLUDE directory root(s). Relative path files will be tried relative to this directory (or each directory in the given directory list, searched from left to right). Multiple directory roots may be specified in the form of a list. A directory root list is separated by semi-colons when on Windows, or colons when on Linux. The * character can be used as a placeholder for the current text of this option (e.g., when you want to add extra directories either before or after the current ones without having to specify the whole thing again.) Since v9.56 two special placeholders may be used in the directory list specification: ?1 which refers to the path of the current main file, and ?2 which refers to the path of the current (parent) file. Since the main file is now searched last, and the current (parent) file's path is no longer searched by default, to get the same behavior as was the default in previous versions, you should run this to convert the current path to the new format but with compatible behavior: *-i.;?1;?2;* -w* (for Linux, use *-i.:?1:?2:**

		-*w*) Since v9.58 FOSSIL source control management users can use the ?F (case-insensitive) placeholder to specify the root directory of the repository. This allows for truly portable installations of your code base. Since v9.61 users can use the ?A (case-insensitive) placeholder to specify the assumed root directory, which must contain a filename named _asm_ (lowercase in Linux) of zero length, even. This allows for truly portable installations of your code base. This switch does not affect absolute path file definitions. Both the INCLUDE and the IF(N)EXISTS directives are affected by this switch.
-L[±]	-L+	Create a *.LST file (one for each file assembled).
-LC[±]	-LC+	List any conditional directives fully (the directives only, not the contents in between), even when they are False. This is the new behavior of ASM11 as of v1.81 but the option is there for those few that liked the old way.
-LLnum	-LL19	Define the maximum recognizable **L**abel **L**ength from the legacy 19 characters up to an absolute maximum of 50 characters. See also the directive **#MAXLABEL**.
-LS[±]	-LS-	Create a *.SYM symbol list (one for each file assembled). May be useful for debuggers that do not support the P&E map file format.
-LSx	-LSS	x may be either S (default) for simple SYM file, E for EM11/Shadow11 SYM format, or N for NoICE SYM format.
-M[±]	-M+	Create a *.MAP (one for each file assembled). *.MAP files created may be used with debuggers that support the P&E source-level map file format.
-MT*x*	-MTP	Specifies type of MAP file to be generated (if **-M+** in effect): **-MTA** : Generate parsable ASCII map file **-MTP** : Generate P&E-style map file
-O[±]	-O+	Enables these four warnings: 'S19 overlap', 'RMB overlap', 'Violation of MEMORY directive', and 'Violation of VARIABLE directive'.
-P[±]	-P+	When on it tells the assembler to stop after Pass 1 if there were any errors. Provides for faster overall assembly process and less confusion by irrelevant side errors of Pass 2. Warnings alone never stop in Pass 1.
-Q[±]	-Q-	Specifies quiet run (no output) when redirecting to an error file (DOS only). Useful for IDEs that call ASM11 and don't want to have their display messed up. Beginning with v2.07, this option can also be used to suppress all output from #Message directives.

-R*n*	-R74	Specifies maximum length of S-record files. The length count *n* includes all characters in an S-record, including the leading «S» and record type, but not the CR/LF line terminator. Minimum value is 12 (for one object byte per record) while maximum is 250 (120 object bytes per record).
-R.*ext*	-R.ASM	Specifies the default extension to assume for source files specified on the command line, which do not directly specify an extension.
-REL[±]	-REL+	Allows generation of *«BRA/BSR instead of JMP/JSR»* optimization warnings when enabled. (See also OPTRELON/OPTRELOFF)
-RTS[±]	-RTS-	Allows generation of *«JSR followed by RTS»* subroutine call optimization warnings when enabled. (See also OPTRTSON/OPTRTSOFF)
-S[±]	-S+	Generate *.S19 object file (one for each file assembled).
-SH[±]	-SH-	Include dummy «S0» record (header) in object file (only if **-S+**).
-S9[±]	-S9+	This option can be used to turn off generation of the final S9 record found by default in all S19 files. This may be useful when assembling code in parts that will be combined with other S19 files. Since you only need a single S9 record in the final S19 file, you can use this option to not produce S9 records for all but one of the files that will be merged together to produce a single object file with a single S9 record. *This option cannot be saved with the -W switch.* *Example: Application and bootloader merging. Assuming you merge the first with the second (in that order), the bootloader should be assembled as usual, and the application with the -S9 option in effect.*
-SP[±]	-SP-	When enabled, the operand part of an instruction is stripped of spaces before parsing. In this case, possible comments must begin with semi-colon. (See also SPACESON/SPACESOFF)
-T[±]	-T-	Makes all errors look like Borland errors (useful to fool certain third-party IDEs).
-T*n*	-T8	Specifies tab field width to use in *.LST files and object code strings. Tab characters embedded in the source file are converted to spaces in the object code strings, and in the listing file such that columns are aligned to 1 + every *n*th character.
-Ux		Define default OUTPUT directory. If this option is defined, all produced files will end up in this directory, regardless of where the source file is located. When this option is undefined (no path given), produced files will end up in the same directory as the primary source file. *Not available in the DOS version.*
-X[±]	-X+	Allow recognition of extra, non-68HC11-standard mnemonics in source files. (See also EXTRAON/EXTRAOFF)
-WRN[±]	-WRN+	Enables or disables the display of all warnings. When enabled, only warnings

		that aren't disabled individually will be generated. When disabled, it overrides local warning options (such as `-REL` and `-RTS`).
-W **-WW**	`(none)`	Write options specified earlier on the command line to the ASM11 executable (DOS), or ASM11.CFG (Win/Linux). The user-specified options become the default values used by ASM11 in subsequent invocations. *Filespec*(s) on the command line are ignored. Assembly of source files does not take place if this option is specified. `-WW` removes licensing information from generated configuration so that it can be distributed to others.

Source File Pseudo-Opcodes (Pseudo-Instructions)

Pseudo-Op	Description
[label] **ALIGN** *expr*	Case 1. If no label is present, it aligns the current location counter to be a multiple of the given expression. Case 2. If a label is present it aligns the value of that label to be a multiple of the given expression. (In this case, however, it does nothing to the current location counter.) It issues an error if the label is not already defined. **COMPATIBILITY ISSUE WITH VERSIONS PRIOR TO 8.30:** Prior to version 8.30, the optional label would be assigned the current location counter value after the alignment. The label could not be defined earlier, or you would get an error. With v8.30 and later, you get an error if the label is not already defined by the time `ALIGN` is reached because the new behavior requires a previous definition so it can align the existing value of the label. This makes it easy to catch all incompatible `ALIGN` statements written for the previous version(s). If you get an error, simply move the label after the `ALIGN` statement.
DB *string*|expr[,...]	Define Byte(s). *expr* may be a constant numeric, a label reference, an expression, or a string. **DB** encodes a single byte in the object file at the current location counter for each *expr* encountered (using the LSB of the result) or one byte for each character in *string*s.
DS *blocksize*	Define Storage. The assembler's location counter is incremented by *blocksize*. Forward references not allowed. No code is generated.
DW *expr*[,...]	Define Word(s). *expr* may be a constant numeric, a label or an expression. *expr* is always interpreted as a word (16-bit) quantity, and is stored in the object file at the current location counter, high byte followed by low byte.
END [*expr*]	Provided for compatibility. The END directive cannot be used to terminate assembly; ASM11 always processes the source file to the

	end of file. If *expr* is specified, the word result is encoded in the S9 record of the object file.
[Label] **ENDM**	Ends definition of a macro. The optional *label* is defined prior to the ENDM processing, which means it is in the same scope as the rest of the macro.
label **DEF** *expr[,size]*	Assigns a DEFault value to a label. *In other words, this is a conditional **EQU**. It only assigns the label if the label is currently undefined.* `LABEL DEF EXPR` is equivalent to: `#IFNDEF LABEL` `LABEL EQU EXPR` `#ENDIF` Note: The value that appears in the listing file is the actual new value of the label, which may be different from the value of the expression, since the assignment may not occur.
label **EQU** *expr[,size]*	Assigns the value of *expr* to *label*. See also **EXP** and **SET**
label **EXP** *expr[,size]*	Assigns the value of *expr* to *label*. This is similar to **EQU** but with the following difference: Labels defined thus will be included in the .EXP file as regular **SET**s. This effectively allows exporting symbols for use from other source files. It makes it possible to give only object code to others along with the produced .EXP file so that they can «link» the object to their source. Found in versions 1.84b+ but will be honored only in 1.85+
FCB *string|expr*[,...]	Form Constant Byte(s). Same as **DB**.
FCC *string|expr*[,...]	Form Constant Character(s). Same as **DB**.
FCS *string|expr*[,...]	Form Constant String. Similar to **FCC**, but it automatically adds a terminating null (0) byte to the end of the string defined (for ASCIZ strings).
FDB *expr*[,...]	Form Double Byte(s). Same as **DW**.
LONG *expr*[,...]	Form 32-bit long word(s). *expr* may be a constant numeric, a label or an expression. *expr* is always interpreted as a 32-bit quantity, and is stored in the object file at the current location counter in big-endian order.

	Not available in the DOS version.
MacroName **MACRO** *comments ...* **REMACRO** *...*	MACRO begins the definition of a new macro. REMACRO begins the definition of a new macro over a possibly existing same name macro. *Hint: Use #DROP to remove the latest definition, restoring the previous one, if any.* 1. Macros must be defined anytime before they are invoked, and they can be invoked until the end of the current assembly *(for global macros)*, end of current file *(for local macros)*, or until a #DROP directive undefines them, in either global or local case. 2. The body of macros is placed between MACRO and ENDM keywords, and it can contain any text. All that text is associated with the specified *MacroName*, as is. Normal semi-colon beginning comments are copied also. If you want comments to appear only in the macro definition but not in each later expansion of the macro, use double semi-colon (**;;**) for those comments to cause them not to be saved along with the macro. 3. By default, macros are invoked using the **@MacroName[*,parm separator*]** syntax (see #MACRO, #@MACRO, #MCF, and #MCF2 directives). Note: You can also use the **%macro** call syntax (i.e., % prefix, instead of @) to force all macro counters (:MINDEX, :INDEX, :LOOP), *except for :MACRONEST*, for the specific macro to reset, as if you had dropped and recreated the macro. 4. During invocation, the macro name may be followed by a comma and any non-alphanumeric single character (if more characters found, only the first matters). If this *parameter override* option is present, then the character right after the comma will act as a one-time parameter delimiter (just for this macro call. The #PARMS defined delimiter will not be affected.) If the character is a space, it does not require yet another space as field separator between macro name and parameters. 5. The macro may refer to yet undefined labels or macros, as the code or definitions inside it are not truly parsed until the macro is actually used, if at all. 6. The macro is expanded on a line-by-line basis. Each line in the macro body (the text between the macro and endm keywords), is expanded and then assembled, before the next line of the

	macro body is fetched. 7. Conditionals used inside macros are always local to the current macro invocation, *i.e., you cannot open a condition (like `#IFDEF`) inside a macro and then close it (`#ENDIF`) outside the macro.* 8. Local macro names start with the `?` symbol (like it is done with normal local labels). 9. The special local macro named `?` *(just a single question-mark)* is to be used ad-hoc. This one special macro name is automatically dropped (without warning) at each new redefinition. It's useful for quickly defining a temporary macro to be used immediately afterwards, and considered discarded later. For example, an instruction (or series of instructions) with a complex operand expression can be embedded inside a `?` local macro using as parameter the variable part of the expression. This can often make your code more readable (and, editable more easily.) 10. Parameters are passed during invocation in the operand field separated by commas (or whatever delimiter you have defined with the `#PARMS` directive, or the special one-time parameter separator override.) 11. To use a null parameter, just put two delimiters next to each other (e.g., `@MACRO PARM1,,PARM3`). Note: This will work for any delimiter <u>except</u> for space; two or more consecutive spaces – outside a string, of course – are seen by the assembler as one space in the parameter field. Space delimiters can only be used with sequential parameters without gaps in between *(which is good for the majority of cases, but not all)*. If you must know, this is because the assembler trims multiple spaces between fields to locate the operand field. If spaces were allowed to separate null parameters, it would also have to count the spaces from the macro name to the parameter field less one that is required to separate the two fields and possibly less one more that could be used with a "space" parameter override, and since the null parameters could be first in the list of parameters, this would be very confusing, and hard to get it to work correctly (especially since you can't easily count spaces) while also maintaining the desired code formatting. So, **when calling a macro with <u>non-trailing</u> null parameters, make sure the**

	parameter separator is NOT a space (either by default or by override), or you will get incorrect macro expansions (and, depending on what the macro does and how it expands, you may not always get side errors). 12. Macro-local labels must include the string `$$$` *at least once* anywhere inside their name (except at the very beginning), e.g. `Loop$$$` or `Main$$$Loop` 13. Parameter text replaces placeholders anywhere within the body of the macro (label, operation, operand, comment fields) without regard to context. Parameter placeholders are `~0~` thru `~9~` (where `~0~` is reserved for the macro name itself, and `~1~` thru `~9~` for actual parameters.) 14. ~C*n*~ where n is a number from 0 thru 9 is equivalent to ~{n}.{:loop}.1~ 15. ~-C*n*~ where n is a number from 0 thru 9 is equivalent to ~{n}.{:{n}-:loop+}.1~ 16. ~c*n*~ where n is a number from 0 thru 9 is equivalent to ~{n}.{:mloop}.1~ 17. ~-c*n*~ where n is a number from 0 thru 9 is equivalent to ~{n}.{:{n}-:mloop+}.1~ 18. The body of a macro may contain *nested* embedded expressions (in any field, even comments) of the form `{<expr>}`, like one can do with strings, where `<expr>` is any valid expression, normally including some parameter placeholder(s). Expressions are evaluated last, after expansion of parameter placeholders but before the `~n.s.l~` type placeholder (described later). 19. To accommodate indexed mode instruction operands within any one parameter (provided the macro is called with a non-comma parameter separator), you can use the following variations of the placeholders: `~n,~` and `~,n~` *(where n is the number 1 thru 9)* and the comma position (either after or before the number) defines whether we want the part before the index (excluding the comma), or the index itself (including the comma), respectively. For example, the instruction `lda ~1,~+1~,1~` will expand correctly whether parm `~1~` contains an index or not. *(Using the simpler `lda ~1~+1` will not expand as intended, when used with indexed operands, as the `+1` will*

	follow the index, and not the offset before the index.) If no index is within the parameter, `~n,~` is the same as `~n~` while `~,n~` is null. The assembler will pick anything following a possible comma (the first one) within a parameter as being an index (so you could get creative and use the feature for other purposes also). 20. The special placeholder `~#~` returns either a null string or the character `#` if the first parameter's (`~1~`) first character is a `#` (possibly, indicating immediate mode use). With conditional assembly (e.g., `#IFPARM ~#~`) one can treat the `~1~` parameter differently, assuming immediate mode. *21.* Similarly, the placeholder `~#n~` (where n is a number from 1 thru 9, zero also accepted but it is pointless) returns the parameter part after a possible `#` sign, if one is present. This allows getting an immediate mode type parameter in a form (stripped of the `#` symbol) that can be used in expressions (for example, in an `#IF` directive expression). *Note: If no `#` is inside the parameter, `~#n~` is the same as `~n~` alone.* 22. Since one may often call a macro with a non-comma delimiter (such as when a parameter contains a comma in an indexed operand – e.g. `1,x`), a possible chained macro call passing this parameter to another macro, or to self while looping, must use the same parameter delimiter that was used to call the original macro, or else the parameter may not be passed on correctly, or not even as a single parameter. Using the default parameter separator (a comma) from within a macro to call another macro (or self) is problematic in those cases. To solve this problem, two equivalent special placeholders have been introduced. One is the ASCII code 149 [•] (e.g., use the `ALT-7` method *in the numeric keypad* for entry in Win-PCs), and the other is the two-character sequence `\,` (a backslash followed by a comma) which should be possible to type in any editor. Either of these placeholders will be replaced by the same delimiter as the one used for the most recent macro call (either by default or by override), unless there is a new explicit one-time delimiter override (`@macro,`***`char`*** call format). *23.* The special placeholder `~label~` *(case-insensitive)* returns the actual text of a label appearing in the label column of the

	last macro invocation (after expanding possible label embedded {<expr>}). This can be used with 'function-like' macros that need to set a label to a specific value (without having to pass the name of the label as a regular parameter). If no label is used in the same line as the macro invocation, then it returns a null (empty) string. If, however, no label is used with a chained (or nested) macro invocation (a macro invocation occurring from inside a macro) then the text value of ~label~ is not changed from the original macro's. This way, a macro can chain to itself (for looping), or another macro and still have the ~label~ placeholder expand correctly. *Note: The length of the actual text inside ~label~ can be found in the internal variable :label.* 24. The special placeholder ~macro~ *(case-insensitive)* returns the name of the top-level macro call (useful when used inside nested or chained macros). For example, if macro A calls macro B, which then calls macro C, then ~macro~ equals A inside all three macros. 25. The placeholder ~00~ returns the name of the macro calling this macro (i.e. the macro one-level above, or the same macro if calling itself). If at the top-level, ~00~ is the same as ~0~. Useful when combining common functionality macros but need the name of the previous macro calling this one. For example, if macro A calls macro B, which then calls macro C, then ~00~ equals A (when inside A or B) but ~00~ equals B (when inside C). 26. The special placeholder ~self~ *(case-insensitive)* returns the original name of the current macro (useful if you use #RENAME from within the macro and then need to restore the actual name the macro had when entered, using #REMACRO). 27. The special placeholder ~text~ *(case-insensitive)* returns the current temporary text parameter of the current macro. This is an temporary placeholder that remembers its macro-unique value across different macro calls, adding extreme flexibility. You can also use it as temporary text workspace when manipulating regular macro parameters. ~text~ can be changed with MSET, MSWAP, MDEF using zero for the parameter index. The current length of ~text~ can be found in the internal variable :TEXT 28. Similarly, the placeholder ~#text~ (case-insensitive) returns

	the part after a possible # symbol(if one is present). 29. The case-insensitive placeholder ~filename~ returns the current file's filename including the file extension, while ~basename~ returns the filename without extension, and ~path~ returns the full path with filename and extension. The variant starting with m *(for macro)* shows the corresponding filename for where the macro definition is located (~mfilename~ etc.), which is not necessarily in the current file. 30. The placeholder ~@~ is an alias for the full list of placeholders separated by • (starting from ~1~). Useful if you want to pass all parameters to another macro. The sequence produced by ~@~ is: ~1~•~2~•~3~•~4~•~5~•~6~•~7~•~8~•~9~ 31. The placeholder ~@@~ is an alias for the full list of placeholders separated by • but starting from ~2~. Useful if you want to pass the remaining parameters to the same macro when looping (assuming each loop only processes the first parameter, until that becomes null). The sequence produced by ~@@~ is: ~2~•~3~•~4~•~5~•~6~•~7~•~8~•~9~ 32. Trailing parameter separators (commas by default) and trailing commas due to macro expansion of null parameters are automatically removed. This is particularly useful when writing macros, which replace or enhance single-operand instructions. One can write the macro so that it does not require a parameter separator override during invocation, just so it can recognize a possible indexed operand. Example: LDA ~1~,~2~ will work even if ~2~ is null, because the now 'dangling' comma after ~1~ will be automatically removed, preventing an otherwise expected syntax error. Similarly, LDA ~1,~+1~,1~,~2~ will work for the following location pointed to by the expression in parm 1, regardless of the presence of an index in parm 1, parm 2, or at all. 33. An alternative to the above method can be achieved (in most cases) by using the **~[n.p]~** format of the parameter placeholder (where **n** is the parameter number, or the case-insensitive keyword *text*) to extract the p[th] byte of the argument (e.g., if ~1~ contains *my_var,sp* ~[1.1]~ will return *my_var,sp* while ~[1.2]~ will return *my_var+1,sp* but if

	~1~ contains an immediate value such as *#$1234* ~[1.-1]~ will return *#$12* while ~[1.-2]~ will return *#$34* etc. For a 32-bit example, for parm *#$12345678* ~[1.1]~ will return *#$12* while ~[1.-1]~ will return *#$56*. The **p** number can be any integer (positive or negative) but for immediate mode parameters the sign matters, and it only works up to 32-bit if positive (i.e., 1..4), or up to 16-bit if negative (i.e., -1..-2). For immediate mode only, a number above 4 or below –2 will return #0 since that is the effective value. This ~[n.p]~ placeholder makes it particularly easy to deal in a unified way with any parameter, be it immediate, direct, extended, or indexed mode. Care has to be taken to use a negative **p** if working with 16-bit values, however. 34. You can use the **~n[set]i~** format of the parameter placeholder to extract the i-th part of the n-th parameter using the character set [set]. The character set can be given as a string of characters with or without quotes (square brackets, instead). Therefore, quotes can also be part of the character set. Example, ~2[,.]3~ will return the 3rd part of the 2nd parameter, where parts are separated by any instance of comma and dot. (Note: 'n' and 'i' are optional. If 'n' is missing, the value one is assumed. If 'i' is missing, the value one is assumed. Care must be taken not to allow both to be missing, if the character set contains a dot because it will then be interpreted as a ~[n.p]~ placeholder, which is processed earlier.) If the character set contains only a single character, then embedded strings will be skipped over, otherwise characters even inside strings will be matched by the characters in the set. 35. You can use the **~n.s.l~** format of the parameter placeholder (where **n** is the parameter number, or the case-insensitive keyword *text* or *label*, or a constant string enclosed in quotes, **s** is the starting position, or a constant string to search for, and **l** is the needed length, or a constant string to search for but past the s position) to extract only a portion of the text of the corresponding parameter or constant. The first dot is required (to disambiguate from ~n~ type parms) even if nothing follows. The s and l are optional. If s is not entered its value is assumed to be one, so that ~1.~ is the same as ~1~

	alone. If `l` is not entered, its value is the length from `s` to the end of the parameter (i.e., the remaining string). Note: The assembler forces `s` and `l` to always be within the limits of the text length. So, specifying a position past the end of the parameter text will always return the last character. To check for past-of-text, check against the `:nnn` length internal symbol for the specific parameter (e.g., `:1` for parm one). If you need to make `n`, `s`, or `l` the result of an expression you can use `{expr}` (for example: `~1.{:loop}.2~`). SPECIAL CASE: When inside a string, the expression will be evaluated when the string is processed by the assembler, which is after macro expansion of the various placeholders. This means we have lost our chance to expand this placeholder. But, we can use the `\@` instead of quotes for strings inside a macro which contain `~n.s.l~` embedded expressions, and not only those (example: `fcc \@~{PARM}.{FROM}.{LENGTH}~\@` to have it expand correctly. Because of the `\@` the string does not appear as a string yet, and the expressions can be calculated during macro expansion. This way all expressions become simple constants, and the placeholder can be processed. Finally, the `\@` dummy string delimiters are turned into single, double, or back quotes, depending on which of these three doesn't appear in the string at all, making the whole thing a proper string. ***IMPORTANT COMPATIBILITY ISSUE:*** *A couple or so versions compiled prior to 2010/09/24 23:00 used `@@` instead of `\@`. The `@@` was an unfortunate selection of dummy quote delimiter and it had to be replaced with a better one (`\@`) even though it meant possibly causing problems with existing code (hopefully, not that many macros utilizing this feature were written in the few days the feature has been available with the wrong delimiter) because it caused syntax errors in certain cases, e.g. if single character string contained the `@` char (with or without macro parameter expansion), or labels containing `@@` inside their name.* 36. Order of placeholder expansion is: **`~@~`**, **`~@@~`**, **`~label~`**, **`~macro~`**, **`~00~`**, **`~self~`**, **`~text~`**, **`~#~`**, **`~#n~`**, **`~n~`** *(where n = 0..9, in that order)*, **`\,`**, and **`•`**, **`{expression}`**, **`~[n.p]~`**,

	~n.s.l~, **~n[set]i~**, **\@string\@,** and **~Cn~** variations. 37. During macro invocation, any parameter text may contain embedded expressions of the form {<*expr*>}, like one can do with strings, where <expr> is any expression, possibly including some parameter placeholder(s), if already inside a macro. This may be needed in situations where the parameter may be intrepreted incorrectly while used inside the macro. For example, if the * (normally used to indicate 'here', as in BRA *) is passed as a parameter to be used inside the macro, it may have a different value, depending on where it is used. Passing this parameter as {*} is first 'expanded' using the current value, and then passed in the macro as a simple constant. Note: You can also do the same expansion from within the macro, making it *worry-free* for the user of the macro. For example, one of the first macro lines can change * to {*} (using MSET) if the specific parameter is found to have this text. 38. Macros cannot #INCLUDE files, but can 'chain' to one. 39. Macros cannot define other macros. *40*. Macro-embedded macros are not supported. *(Tip: Simple 'embedded macros' can be emulated by using any unused parameters to contain the text of the 'embedded macro'. The MSET keyword can be used from within the macro to 'define' the 'embedded macro' in one or more unused parameters, – each parameter representing a single line of the 'embedded macro' – then use just the relevant placeholders alone wherever you want to expand the 'embedded macro'.)* 41. Macros can 'chain' to self or other macros (with no automatic return). This allows, among other things, for creating loops, making macros very powerful. *42*. Macros can temporarily invoke other macros, and then return back to continue with the original macro. Use the double @ (@@ or %%) notation when calling a macro from within another macro if you want to return back (as opposed to chain to another macro), regardless of macro mode. The default maximum nesting level is 100 (which should be more than adequate for most cases) but it can be changed to as high as 10,000 with the directive #MLimit, or as low as zero, which disables this capability completely. *Note: Prefer using macro*

	chaining over nested macro calling when feasible, or to get a looping effect, as it is more efficient both in terms of memory usage and assembly speed. Tip: To use macros as with some other assemblers, i.e., without having to type `@` prefix, and having a default nesting (rather than 'chaining') behavior, enable the ***#MACRO @@*** *mode (see the relevant section for details) . Macro-chaining will be altogether disabled, however.* 43. To break out of an accidental endless macro loop, press [ESC] on the command-line. *44.* Macro labels may be case-sensitive (depending on `#CaseOn/Off` directives) when defined, but are always case-insensitive when invoked (like normal opcode names). *Tip: A case-sensitive macro definition is important when using the `~0~`, `~00~`, and `~macro~` placeholders to have it correctly match a normal label named the same as the macro, under `#CaseOn` mode.* *45.* Virtually unlimited number of macro definitions *(memory permitting.)* *46.* Virtually unlimited size of each macro *(memory permitting.)* 47. Unlimited number of macro invocations *(all internal macro counters are 32-bit).*

MERROR [text]	Combines an #ERROR directive followed immediately by an MEXIT, which is commonly found in macros. *This can only be used inside macros.*
MEXIT [expr]	Causes an unconditional early exit from a macro expansion. (Normally, used inside a conditional block.) If the optional expression (without any forward references) is present, its value will be placed in the :MEXIT internal variable. If the expression is missing, the current value of :MEXIT will not be changed, allowing for cascaded return values from nested macros.
MSUSPEND **[!]MRESUME**	MSUSPEND can be used only from within a macro (usually once, but since there is no limit, more than once, if needed) to temporarily suspend the execution of the current macro. Suspending a macro preserves the current macro state (parms, counters, etc.) just like nested macros do to protect the parent macro's state, but it allows for code outside any macros to be assembled in place of the MSUSPEND keyword, as if it were part of the macro (except that it is actually assembled outside the macro, so none of the macro-only features can be used, and none of the macro limitations apply – for example, normal use of #INCLUDE is possible, as well as definition of new macros, etc.) This makes it much easier to create *nestable* macros that emulate block structures, than by using two separate macros (one for block begin, and one for block end) and trying to keep them synchronized. *Note: When a nested macro is suspended, <u>all</u> macros leading to the currently executing macro are indirectly suspended as a side effect.* MRESUME can be used only from outside any macros to resume execution of the most recently suspended macro. (!MRESUME produces no error.) You can have several macros in the suspended state, but they can only be resumed in a LIFO order (i.e., stack order). This allows for the creation of nested blocks (like WHILE, FOR, IF, REPEAT, etc.)

commonly found in higher-level languages.

The recursion limit (see `#MLimit`) counts suspended macros also, because these are stacked just like when doing normal nested macro calls.

This `MSUSPEND`/`MRESUME` feature makes it particularly easy to replace pairs of macros (like `FOR` … `ENDFOR`) that normally appear right before and after a code section to create a block structure, with a single macro that does all the required work and simply allows (via the use of the keyword `MSUSPEND`) the inclusion of any arbitrary code in between (i.e., between the macro call and the `MRESUME` keyword).

Simple *nested* example (counts lines of intermediate source code, and issues warning if optional limit is exceeded):

```
                    org       *

CountLines          macro     [Limit][,Description]
                    mset      2, ~@@~
                    #temp     :lineno+1
                    msuspend
                    #temp     :lineno-:temp
                    #Message  Section~2~ spans {:temp} lines
          #ifnb ~1~
          #if :temp > ~1~
                    #Warning  Too many lines (>{~1~})
          #endif
          #endif
                    endm
; To use:
                    @CountLines ,OUTER
                    nop
                    @CountLines ,INNER
                    nop
                    nop
                    mresume
                    nop
                    mresume
```

MSTOP [#ALL#] **MSTOP** [text]	When used inside a macro, it causes an unconditional early termination of all currently executing macros, and regardless of nesting level. (Normally, used inside a conditional block.) When used outside a macro, it causes the most recent suspended macro to stop being suspended. When the optional #ALL# parameter is used, then all nested suspended macros are stopped (become no longer suspended).
MSTR index[,index]*	**MSTR** tests each one of the specified indexed macro parameter text for being a string, and, if not a string, it changes it to one using the appropriate delimiters based on the contents of the parameter text. It is equivalent to the following sequence (but repeated for each specified index **n**): #IFPARM ~**n.**~ #IFNOSTR ~**n.**~ MSET **n,**\@~**n.**~\@ #ENDIF #ENDIF
MSET index[,text] **MSET #** **MSET #'charset'** **MDEF** index[,text] **MSWAP** index,index **MDEL** index **MTRIM** index[,index]*	**MSET** changes the current macro's index-ed parameter to the text that follows, or to null if no text follows. There are many potential uses for this capability (such as using the macro parameters as temporary text variables.) It is particularly useful, however, with macro loops using the MTOP command. A second variation of MSET (added in v8.90) allows to unite all parameters into one, and optionally split back into as many parameters using a user-defined character set given as string. For example, MSET # will unite all current macro parameters into just one (using the currently active macro parameter separator.) MSET #'abc' will first unite alls parms into one and then split into separate parms for every occurrence of characters a, b, or c, while skipping over parenthesized parts, and strings. If the string after # has just one character, this will be eliminated (as it will be treated as the new parm separator.) If it has more than one character, the parm will be split right before the character, and the found character will be the first character of the next parm. This feature allows you to re-arrange the parameters based on a different separator.

	MDEF is similar to `MSET` but it only changes the text of the parameter if the parameter is currently null. This is the same as using `MSET` within an `#IFNOPARM` conditional block. It's useful for setting default macro parameters (normally, at the top of the macro). **MSWAP** simply swaps the text of any two parameters. *(Swapping a parameter with itself has no effect.)* As an example for `MSWAP`, in macros with multiple single operands, you can use it to bring the working operand always in, say, `~1~`, which may be simpler to use than the equivalent `~{:loop}.~` from inside a loop. **MDEL** deletes the current macro's `index`-ed parameter. Note: This is different that using `MSET` without the text parameter to delete the content of a parameter placeholder. `MDEL` deletes the actual parameter location, which means all following parameters will move down one location. For example, `MDEL 1`, will move `~2~` to `~1~`, `~3~` to `~2~`, and so on, for all parameters that follow. **MTRIM** will trim all non-string spaces from the respective parameter(s). *Note: In all cases, `index` is any expression that doesn't contain forward references.*

MREQ ind[,ind]*[:errmsg]	Checks each of the specified macro parameters (separated with commas) for null value (empty). If the parameter is null, an appropriate internal error message is displayed, and the macro expansion is terminated at that point. If the optional errmsg parameter is present (which must follow a colon), this error message will be displayed instead of the default error message. This can be used to specify which macro parameters are required, and print an error message, if these parameters are null. If more than one of the specified parameters are null, the message will repeat for each one of them. You may use MREQ multiple times, perhaps, once for each parameter, so that you can have a unique error displayed for each parameter. *Note: ind is any expression that doesn't contain forward references. errmsg is any text. If ind contains an internal variable (such as :LOOP), it must be enclosed in { ... } because the colon is also used as the beginning of the errmsg.*
MTOP [limit expr]	Causes an immediate *unconditional* jump to the top line of the current macro, while incrementing the :LOOP counter. It can be used either alone or within conditionals. The advantage to using MTOP over @~0~ (a macro call to self) is that whatever parameters were passed in the macro do not need to be specified again as the macro is never exited. Also, no counters are incremented, except for :LOOP. This means, however, that $$$ based labels *(which are unique to a macro invocation)* are still in the same scope as before the MTOP command since no new macro has been invoked. If the optional limiting expression (containing only non-forward references) is present, its value will be compared to the :LOOP counter and MTOP will execute only if the current value of :LOOP is less than the value of the expression. Example (shift word right one or more times): `lsr.w macro Address[,Count]` `mdef 2,1 ;default Count=1` `lsr ~1~`

	`` ror ~1,~+1~,1~ mtop ~2~ endm ``` As another example, an expression like the one that follows can be used to loop while the next parameter is not null: `mtop :loop+:{:loop+1}`

MDO [start expr]
MLOOP [limit expr]

MDO and MLOOP work together to form a local DO ... LOOP inside a macro. *Note: MDO and MLOOP cannot be nested because MLOOP always matches the most recent MDO of the current macro.*

MDO simply marks the current line (i.e., the line containing the MDO keyword) as the beginning of a local loop, and (re)initializes the :MLOOP counter to one (1), or to the value of the non-forward expression, if one is present.

MLOOP causes an immediate *unconditional* jump to the line following the most recent MDO keyword, while incrementing the :MLOOP counter (not to be confused with the :MACROLOOP or :LOOP counter). If no MDO was used up to this point in the macro, MLOOP jumps to the top of the current macro (just like MTOP would), but it only affects the :MLOOP counter (whereas MTOP only affects the :LOOP counter).

If the optional limiting expression (containing only non-forward references) is present, its value will be compared to the :MLOOP counter and MLOOP will execute only if the current value of :MLOOP is less than the value of the expression. Example (multi-byte addition):

```
add.m     macro     Op1,Op2,Ans[,Size]
          mdef      4,1       ;default size = 1
          psha

          mdo
          ldaa      ~1,~+{~4~-:mloop}~,1~
#if :mloop = 1
          adda      ~2,~+{~4~-:mloop}~,2~
#else
          adca      ~2,~+{~4~-:mloop}~,2~
#endif
          staa      ~3,~+{~4~-:mloop}~,3~
          mloop     ~4~

          pula
          endm
```

As another example, an expression like the one that follows can be

	used to loop while the next parameter is not null: `mloop :mloop+:{:mloop+1}`
label **NEXP** *symbol[,expr]*	Assigns the current value of *symbol* to *label* as if with EXP. Then, it increments the value of *symbol* by one (as if with SET) or, if the optional expression is present, by the value of that expression. Useful for defining a series of symbols based on a common starting value. Note: *symbol* is a single label and not an expression. See also **NEXT, SETN**
[label] **NEXT** *symbol[,expr]*	Assigns the current value of *symbol* to *label* as if with EQU. Then, it increments the value of *symbol* by one (as if with SET) or, if the optional expression is present, by the value of that expression. Useful for defining a series of symbols based on a common starting value. Note: *symbol* is a single label and not an expression. Since v9.41, a special case of NEXT is when the *label* to the left is missing. In that case, NEXT is used as an anonymous placeholder that simply increments the *symbol* to the right, as usual. See also **NEXP, SETN**
ORG *[[s19_expr],]expr*	Sets the assembler's location counter for the active segment. Code generated after this directive will be assembled starting at the location specified by *expr*. If *s19_expr* is present, then the S19 file runs with an offset from the actual location counter. This allows for different segments of code to be assembled at the same physical address but, obviously, be placed in different addresses in the loadable S19 file. The current offset is available in the :OFFSET internal symbol. To cancel any offsets without changing the current position, simply give ORG followed by a single comma without any expressions.
label **PROC** *[label]* **ENDP**	PROC first advances the @@ local label counter, and then it assigns the value of the program counter (*) to *label*. This allows using symbols locally for a specific section of code (e.g., a subroutine). The symbol to the left of PROC is always in the new scope. The name of the symbol is stored in the ~procname~ macro placeholder. Each time PROC (or #PROC) is encountered, the assembler increments an internal 32-bit local symbol counter.

	Symbols containing `@@` anywhere inside their name (except at the very beginning) *at least once* (for example, `Loop@@`) will have the `@@` part replaced with a special control character (different from what is used with macro local `$$$`) and the current value of the internal local symbol counter (similar to `$$$` with macro local labels). Up until a `PROC` or `#PROC` is encountered in the program, the `@@` is not treated specially (i.e., the `@@` is not converted to a special number). This makes this feature compatible with code written prior to its introduction. The current value of the corresponding internal counter can be found in the internal symbol `:PROC` while the maximum proc number can be found in the internal symbol `:MAXPROC` `ENDP` is optional and marks the end of the corresponding `PROC`. Its use allows one to nest procs (e.g., for code coherency as when keeping a subroutine close to the actual point of use). The optional *`label`* is defined prior to the `ENDP` processing, which means it is in the same scope as the rest of the proc. See also **#PROC** and **#ENDP**
RMB *blocksize*	Reserve Memory Byte(s). Same as **DS**.
label **SET** *expr[,size]*	Assigns the value of *`expr`* to *`label`* even if label is already defined with a different value. This is similar to **EQU** but allows making multiple re-definitions. The value set will be used until another **SET** pseudo-instruction or to the end of the assembly process. *Warning:* Careless, or simply wrong use of this directive can lead to multiple side errors or warnings (please note this is a two-pass assembler). Using a forward `SET` defined symbol may lead to problems, as the value used will be the one from the last `SET` definition, which is not necessarily the one we want. Correct behavior is guaranteed if any symbols re-defined with `SET` are used only after each new re-definition, otherwise, the first reference in Pass 2 will use the value from the last re-definition in Pass 1. *Example of wrong use:* `1.  lda  #Value  ;we expect 123, actual is 234` `2. Value  equ  123`

	... 3. lda #Value ;we expect 234, actual is 123 4. Value set 234 Value in line 1 will be 234 (the last known value from Pass 1) while Value in line 3 will be 123 (most recent value in current Pass 2). *Example of correct use:* 1. Value equ 123 2. lda #Value ;we expect 123, actual is 123 ... 3. Value set 234 4. lda #Value ;we expect 234, actual is 234 See also **EXP** and **EQU**
label **SETN** *symbol[,expr]*	Assigns the current value of *symbol* to *label* as if with SET. Then, it increments the value of *symbol* by one (as if with SET) or, if the optional expression is present, by the value of that expression. Useful for (re-)defining a series of symbols based on a common starting value. Note: *symbol* is a single label and not an expression. See also **NEXP, NEXT**

Source File Processing Directives

- All processing directives must be prefixed with a **$** or **#** character. ASM11 will recognize either character as the start of a processing directive.
- If a directive has a corresponding command-line option, the directive in the source file will override the command line directive at the point in which the source file directive is encountered.
- `[text]` will be trimmed of duplicate spaces. To have more than one consecutive space display, use the Alt-255 character, as many times as needed.

Directive	Description
#AIS *[symbol]*	`#AIS` checks the current value of the `:SP` internal variable against the most recent `AIS` instruction's value, and issues a warning if the two numbers do NOT differ by the exact value in the symbol *(note: a plain symbol, not an expression)*, indicating a possible stack frame definition error (assuming correct placement of the relevant directives). The warning also shows the correct `AIS` instruction that is required to correct the problem. This directive makes it very easy to correct the numeric value in `GETX/GETY` instructions to match the following stack frame definition (normally made using the internal `::` symbol in the various `#SPAUTO` modes, and the `next`/`setn` method for defining records/structures.) This is useful to prevent having to define the stack frame before the `GETX/GETY` instruction using a one-based starting offset just so you can use a label with `GETX/GETY` and then having to re-define it for dynamic assignment of offsets based on the current `:SP`. If, however, a symbol is not specified, then this directive simply resets the value of the `:AIS` internal symbol to the current `:SP` value difference. This can be used when no actual `AIS` instruction is used (for example, a series of `PSHx` instructions are used, but we want to use the `:AIS` variable later on to de-allocate any local variables.)

	The associated `:AIS` *symbol returns the difference between the current* `:SP` *and the value saved during the most recent* `#AIS` *directive. This can be used to de-allocate just the number of stack bytes that are still left on the stack between the two points in your source. This is only meant for use in* `#SPAUTO` *modes, which automatically adjusts the current value of the* `:SP` *internal symbol.* `#PUSH` and `#PULL` will save/restore the value of this setting.
#CASEOFF	When **#CASEOFF** is in effect, all symbol references that follow are converted to uppercase internally before they are searched for or placed in the symbol table. (Debug and DEBUG are the same symbol.) Equivalent to the **–C–** command line option.
#CASEON	When **#CASEON** is in effect, symbol references are NOT internally converted to uppercase before they are searched for or placed in the symbol table. (Debug and DEBUG are two different symbols.) Equivalent to the **–C+** command line option.
#CRC *expr*	The two CRCs (user and S19) maintained by the assembler are 16-bit each, and they are updated only during PASS2 by each produced user code/data byte that is put into the S19 file. The starting CRC value for both CRCs is zero. With this directive you can alter the user CRC value at any time (either before the very first byte of code/data to produce a different CRC for the same firmware, or several times in between to skip certain volatile sections, for example). The computed CRCs are available by accessing the internal symbols **:CRC** and **:S19CRC** The formula used for the 16-bit CRC calculation is very simple to be easily implemented even in tiny bootloaders: **16BitCRC := 16BitCRC + 16BitAddress*8BitData** `:S19CRC` is mostly useful with the `END` directive (`END :S19CRC`) as it is not affected by the `#CRC` directive. An S19 loader can check the

	overall integrity of the S19 file. `:CRC`, on the other hand, is mostly useful for checking code after it has been loaded into the MCU, at each reset, for example. Please note that for both CRCs all `$00` bytes do not affect the calculation while, for the user CRC only (`:CRC`), all `$FF` bytes are intentionally skipped. This allows for the CRC in an S19 file (which does not necessarily fill a contiguous block of memory) to match the CRC computed by the MCU over a complete block of memory without the MCU bootloader knowing in advance the actual addresses used within that block, provided any unused bytes are in the erased state. *As a side effect, however, any $00->$FF or $FF->$00 alterations in the file cannot be detected with the user CRC.*
#CYCLES *[expr]*	First, the optional expression is calculated using the current values of any internal symbols. Then, the current value of `:CYCLES` is copied to `:OCYCLES`. Finally, the internal `:CYCLES` counter is set to zero (if the optional expression is missing), or to any arbitrary value (the result of the expression). *This directive can also be used inside macros to restore the cycle counter of surrounding code, if the macro cycles should be counted in a special way, or not at all.*
#DATA	Activation of the DATA segment. Default starting value is $103F (the CONFIG register).
#[!]DROP `macro[,macro]*`	Undefines one or more macros. If a macro is not currently defined, a warning will be issued (to protect from possible typing errors) unless the `!DROP` form is used. To drop all macros (global and local) with a single command, use `*` (asterisk) in place of the macro name. *There is no warning if no macros found.* To drop all local macros (for the current file only) with a single command, use `?*` (question mark followed by asterisk) in place of the macro name. *There is no warning if no local macros found.*

	If used from inside a macro, and that macro is dropped, the macro will terminate at that point. The rest of the macro will not be processed. The special macro named `?` (just a single question-mark) is to be used ad-hoc, and it is automatically dropped (without warning) at each new redefinition. You may also drop it with `#DROP` but only need to do so if you want to force errors in later use of the macro, so you can easily locate them. You cannot drop macros that are currently active above the current macro level (e.g. nested macros leading to current one.)
#EEPROM	Activation of the EEPROM segment. Default starting value is $B600.
#EJECT	See #PAGE
#ELSE [IFxxx]	When used in conjunction with conditional assembly directives (**#IF, #IF[N]DEF, $IF[N]Z, #IFMAIN, #IFINCLUDED**, etc.), code following the **#ELSE** directive is assembled if the conditional it is paired with evaluates to a not-true result. Optionally, you can follow with another IF directive (of any kind) to create a 'chained' condition check, like: `#IF … #ELSE IF … #ELSE IF … #ELSE … #ENDIF` The optional IF should not start with a # or $ directive symbol but it should be separated with at least one space.
#ENDIF	Marks the end of a conditional-assembly block. Conditional assembly statements may be nested if they are properly blocked with **#ENDIF** directives.
#ERROR [*text*]	When encountered in the source, the assembler issues a error message in the same form as internally-generated errors, using the *text* specified, prefixed with «USER: »
#EXIT [*expr*]	If no expression is present, it immediately exits the current **#INCLUDE** file. *(Does nothing if used inside a main file.)* If the optional expression is present (normally though, this might be just a single label), the exit occurs only if the expression is defined (as if when checked with `#IFDEF`). This can be used in the top of `#INCLUDE` files, like so:

	`#EXIT _COMMON_` `_COMMON_` In this example, the first time this file is included, the symbol `_COMMON_` is undefined, so the `#EXIT` is ignored. Consequent times this file is included, it exits upon hitting the `#EXIT` directive. *Note: Due to how `#INCLUDE` files are counted internally, and there being a limit on how many total files you can `#INCLUDE`, it's better when working with larger projects that you do not `#INCLUDE` a file at all when already processed, rather than `#INCLUDE` it and `#EXIT` it. (See also `#USES`)*
#EXTRAOFF	Disables recognition of ASM11's extended instruction set for source lines that follow this directive. Equivalent to the **–X–** command line option.
#EXTRAON	Enables recognition of ASM11's extended instruction set for source lines that follow this directive. Equivalent to the **–X+** command line option.
#[!]EXPORT `symbol[,symbol]*`	Export one or more symbols (as if with `EXP`). File-local symbols cannot be exported. If a symbol is not currently defined, a warning will be issued. The optional ! eliminates warnings.
#FATAL [*text*]	Similar to the **#ERROR** directive, but generates an assembler fatal error message and terminates the current file assembly (processing will continue with possible further files in the command line supplied file list).
#HOMEDIR [*path*]	Makes the specified *path* the current home directory. Although this cannot affect where any output files will go, it does make a difference on where any following *relative* **#INCLUDE** files will be searched. Relative file path specifications will now be relative to the directory specified by the **#HOMEDIR** directive, including any relative **#INCLUDE** references in nested include files. If [*path*] is missing, the original main file path is restored.
#[!]IF *expr1 cond expr2*	Evaluates *expr1* and *expr2* (which may be any valid ASM11 expression) and compares them using the specified *cond* conditional operator. If the condition is true, the code following the **#IF** operator is assembled, up to its matching **#ELSE** or **#ENDIF** directive.

	Cond may be any one of: **< <= = >= > <>** The condition is always evaluated using unsigned arithmetic. If a symbol referenced in *expr1* or *expr2* is not defined, the statement will always evaluate as false. At least one space must embrace *cond* on each side.
#IF[N]DEF *expr[|expr]**	Attempts to evaluate *expr*, and if successful, assembles the code that follows, up to the matching **#ELSE** or **#ENDIF** directive. This directive is used to test if a specified symbol has been defined. Symbol(s) referenced in *expr* must be defined before the directive for the result to evaluate true (e.g., forward references will evaluate as false). **#IFDEF** without an expr following will always evaluate to False. You can have multiple unrelated expressions separated by ASCII character 179 (looks like a pipe symbol) which are ORed (**#IFDEF**) or ANDed (**#IFNDEF**) to decide whether the result will be true or false.
#IFEXISTS *fpath*	It checks for the existence of the file specified by *fpath* (using the same rules as those used for **#INCLUDE** directives) and assembles the code that follows if the specified *fpath* exists.
#IFINCLUDED	Assembles the code which follows if the file containing this directive is a file used in an INCLUDE directive of a higher-level file (regardless of nesting level). See also **#IFMAIN**
#IFMAIN	Assembles the code that follows if the file containing this directive is the main (primary) file being assembled. See also **#IFINCLUDED**.

#IFMDEF *macro* **#IFNOMDEF** *macro*	**#IFMDEF** checks if the specified macro exists, and if so, assembles the code that follows, up to the matching **#ELSE** or **#ENDIF** directive. This directive is used to test if the specified macro has been defined. **#IFNOMDEF** does the opposite check.
#IFPARM *text [= text]* **#IFPARM** *text [== text]* **#IFNOPARM** *text [= text]* **#IFNOPARM** *text [== text]* *Aliases:* **#IFB** same as #IFNOPARM **#IFNB** same as #IFPARM	Normally used inside macros. If text is non-blank, assembles the code that follows, up to the matching **#ELSE** or **#ENDIF** directive. This directive is used to test if a specified macro parameter has been defined. **#IFPARM** without text following (after macro expansion) will always evaluate to False. `text` is usually a parameter placeholder (e.g., `~1~`). You can also make a case-insensitive (using the = sign) or case-sensitive (using the == sign) comparison of the parameter to a specific `text` string (with or without quotes, depending on your intent) by separating the two text strings with an 'equals' (=), or double-equals (==) sign, depending on the desired case-sensitivity. For example, `#IFPARM ~1~ = *` tests if parameter one is a plain asterisk (normally used to indicate the current location pointer.) **#IFNOPARM** performs the opposite test.
#IFSPAUTO	Assembles the code that follows, up to the matching **#ELSE** or **#ENDIF** directive, if the assembler is currently in `#SPAUTO` *(automatic SP adjustment)* mode. See also **#SPAUTO #SP**
#IFSTR *text* **#IFNOSTR** *text*	Normally used inside a macro. If text is a quoted string, assembles the code that follows, up to the matching **#ELSE** or **#ENDIF** directive. This directive is used to test if a specified macro parameter is a string. **#IFSTR** without text following (after macro expansion) will always evaluate to False. `text` is usually a parameter placeholder (e.g., `~1~`). **#IFNOSTR** performs the opposite test.
#IFNUM *text* **#IFNONUM** *text*	Normally used inside a macro. If text represents a number, assembles the code that follows, up to the matching **#ELSE** or **#ENDIF** directive. This directive is used to test if a specified macro parameter is a number. **#IFNUM** without text following (after macro expansion) will always evaluate to False. `text` is usually a parameter placeholder (e.g., `~1~`). **#IFNONUM** performs the opposite test.
#INCLUDE *fpath*	Includes the specified *fpath* file in the assembly stream, as if the

#USES *fpath*	contents of the file were physically present in the source at the point where the **#INCLUDE** directive is encountered. #INCLUDE's may be nested, up to 100 or 125 levels (the main source file counts as one level). Relative *fpath* specifications are always referenced to the directory in which the main source file resides, including any relative **#INCLUDE** *fpath* references in nested include files. **#USES** is an alternative, slightly different method to include a file. It will #INCLUDE the file specified (using the same file-finding rules as #INCLUDE) but only if the same file path has not been included (via #INCLUDE or #USES) at least once, already. #USES is useful for creating #INCLUDE file dependencies (normally, from a higher level to a lower level – *e.g., an analog temperature sensor driver module #USES the A/D driver module, but not the other way around*). This allows directly #USING (an alias for #USES) only the module of interest in your application, and it should take care to use whatever other modules it requires (in a recursive sort of way). If another included module in the same application #USES the same lower-level module, it will not be included a second time. This is similar to the common `#IFNDEF _MODULE_` `_MODULE_` `  ...your module code goes here...` `#ENDIF` technique used to prevent multiple inclusions of the same file, but only have it included the first time it is referenced. Normally, the #IFNDEF … #ENDIF block is found inside the file, meaning the assembler must enter the file before it 'knows' it doesn't need it. The advantage with #USES, however, is (1) that you do not need a specific symbol definition for each file, and (2) you never enter an already included file (which would use up a sometimes precious file count towards the maximum number of #INCLUDE files.) Bi-directional, or circular co-dependencies (e.g., file A depends on file B, while file B depends on A) are possible in some cases, and then they require some extra attention in the respective files' internal organization, or it could not work as you might have

	expected, and leave you confused by 'spurious' errors. In general though, you should try to avoid them. Also, you cannot use `#USES` in place of `#INCLUDE` for modules that must be included multiple times (e.g., including the same SCI driver module, once for each hardware SCI available), *although you could use `#USES` to include a file that itself does `#INCLUDE` the same file multiple times.* Note: The assembler will only generate a standard error (not an assembly-terminating fatal error) if a file specified in a **#INCLUDE** (or **#USES**) directive is not found. The **#IFEXISTS** and **#IFNEXISTS** directives may be used in conjunction with **#FATAL** if termination of assembly is desired under such conditions.
#IFNDEF *expr*	Evaluates *expr* and assembles the code that follows if the expression could NOT be evaluated, usually as the result of a reference to an undefined symbol. This directive is the functional opposite of the **#IFDEF** directive.
#IFNEXISTS *fpath*	The opposite of **#IFEXISTS**; code following this directive is assembled if the specified *fpath* does NOT exist. As of version 1.61, the -Ix path will also be searched to determine whether a file exists or not.
#IFNZ *expr*	Evaluates *expr* and assembles the code that follows if the expression evaluates to a non-zero value. **#IFNZ** always evaluates to false if *expr* references undefined or forward-defined symbols.
#IFZ *expr*	Evaluates *expr* and assembles the code that follows if the expression is equal to zero. **#IFZ** always evaluates to false if *expr* references undefined or forward-defined symbols.
#IFTOS *expr*	If top-of-stack evaluates *expr*`+:SP` (`+:SP` is implied) and assembles the code that follows if the expression is equal to one (when in `#SP[AUTO]` modes), or zero (when in `#SP1` mode), i.e., expression points to top-of-stack in all modes. **#IFTOS** always evaluates to false if *expr* references undefined or forward-defined symbols. Useful mostly in `#SP[AUTO]` modes.
#LISTOFF **#NOLIST**	Turns off generation of source and object data in the *.LST file for all lines which follow this directive. Useful for excluding the contents of **#INCLUDE** files in the *.LST file. This directive is not

	shown in the *.LST file.
#LISTON **#LIST**	Enables generation of source and object data in the *.LST file for the source code following this directive. Has no effect if list file generation is disabled (**-L-** command line option in effect). This directive is not shown in the *.LST file if the listing was turned off just prior to it.

#MACRO [@@] **#MCF [@@]** **#MCF2 [@@]** **#@MACRO [@@]**	**#MACRO** tells the assembler to treat unknown assembly language operations as possible macros. Normal instructions (including the built-in macro instructions) have priority over macros, so macros named the same as active built-in operations can only be called with the `@` prefix. In effect, when in this mode, the assembler automatically adds the `@` symbol if an unknown operation is found to be a macro name. In this mode, one can invoke macros either way, with or without the `@` prefix, but instructions have priority over same name macros. *Note: To avoid problems, all macros should internally use the `@macro` syntax so they can be properly expanded regardless of mode.* **#MCF** ("**M**acros **C**ome **F**irst") is similar to `#MACRO` (i.e., no `@` prefix is required for calling macros) but in this case macros have priority over same-name instructions but only when called from outside any macros. Macro chaining (i.e., jumping to a macro from inside a macro) is still only possible using the `@` prefix when a macro name collides with an active instruction name. So, using this mode is 100% compatible with macros written before this mode was introduced and does not require editing macros to use the `!instruction` format mentioned next. If you're in `#MCF` mode, and you want to temporarily give priority to a real instruction (without changing to `#Macro` or `#@Macro` mode), you must prefix it with a **!** (exclamation point.) The `#MCF` mode is most useful when you want to override the functionality of any internal instruction with something more involved (a macro), as for example, when porting code from another CPU with similar instructions but different functionality (e.g. `LDX` in 68HC11 is a word operation, and it may compile without errors in the 68HC[S]08 but with incorrect operation as it will not affect the full HX register). *I do not recommend casual use of this mode as it may make the source code totally misleading (if instructions which are now possibly macros aren't what they seem but something completely different.)* **#MCF2** is almost the same as `#MCF` but it doesn't have the

	restriction where macros named the same as instructions require the `@macro` format from within macros. This is the most 'dangerous' of all available modes, since it is always the macro which has precedence. If you need to be certain you use a real instruction and not a possible macro with the same name, you MUST use the `!instruction` format. **#@MACRO** turns off this option. This is the default setting when a new assembly begins. In this mode, you can only invoke macros with the `@` prefix. This is the recommended mode for most normal applications. *Hint: The macro is normally invoked as an instruction, which means its name must appear after column one. Regardless of the current macro mode, when a macro call is made using the default `@macro` (or `%macro`) format, its invocation can start even in column one, since it can't ever be a symbol that starts with one of these two characters [`@` and `%`].* Note: If the optional `@@` parameter is provided to any of the four directives mentioned above, macro chaining is effectively disabled, and any otherwise 'chained' calls now become truly nested calls (as if the `@@macro` format is used at all times a macro is called). WARNING: Macros written based on the default 'chain' behavior may no longer operate the same (since non-`@@` macro calls include an implied following `mexit`). To simulate the same behavior, when the `@@` option is active, make sure you add an `MEXIT` command after each otherwise 'chained' macro call. By the way, this will make the macro work the same way regardless of the `@@` sub-mode being in effect or not. When the `@@` sub-mode is in effect, you still need to observe the various calling methods based on which of the three macro modes you're in. To cancel the `@@` sub-mode, simply give any of these directives without it.
#MAXLABEL number	Define the maximum recognizable **L**abel **L**ength from the legacy 19 characters up to an absolute maximum of 50 characters. See also the command-line option **-LL**.

	`#PUSH` and `#PULL` will save/restore the value of this setting.
#[!]MEXPORT `macro[,macro]*`	Export one or more macros in the EXP file (if one is produced). File-local macros cannot be exported. If a macro is not currently defined, a warning will be issued. The optional ! eliminates warnings.
#MLIMIT *[expr]*	Sets the maximum macro nesting limit to the value of the optional expression. If no expression follows the default value of 100 is used. This value should be more than adequate for nearly all cases. Minimum value is zero (which practically disables macro call nesting). Maximum is 10000 (ten thousand). *Note: Macro nesting uses extra memory during assembly. You should avoid using macro nesting if the same functionality can be achieved by using macro chaining, or even the most efficient simple looping (`MTOP` instruction).*
#MLISTOFF **#NOMLIST**	Turns off generation of source and object data in the *.LST file for all macro body lines which follow this directive. Useful for excluding the body of macros in the *.LST file.
#MLISTON **#MLIST**	Enables generation of source and object data in the *.LST file for all macro body lines following this directive. Has no effect if list file generation is disabled (**-L-** command line option in effect). This is the default setting.
#HIDEMACROS **#SHOWMACROS**	*Note: These two directives work only when the `-LC-` (List Conditionals = OFF) command-line option is in effect.* `#HideMacros` treats all macro-specific keywords (the `@macro` call, `mexit`, `mtop`, `endm`) the same as 'conditional' directives only for the purposes of display in the listing. So, when `-LC-` is in effect, they won't appear in the `*.LST` file at all. This leaves only the expanded macro contents. When this directive is in effect, it is no longer possible to know where a macro begins or ends, or how many times it iterates itself. *Note: The corresponding macro definitions will not display at all, regardless of the `-LC` mode.* `#ShowMacros` (re-)enables normal display. The default setting when a new assemby begins is `#SHOWMACROS`.

	#PUSH and #PULL will save/restore the value of this setting.
#MAPOFF	Suppresses generation of source-line information in the *.MAP file for the code following this directive. Symbols which are defined following this directive are still included in the *.MAP file.
#MAPON	Enables generation of source-line information in the *.MAP file for the code following this directive. **#MAPON** is the default state when assembly is started when map file generation is enabled (**-M+** command line option).
#MEMORY addr1 [addr2] **#MEMORY** #OFF#	Maps a memory location (or range, if addr2 is also supplied) of object code and/or data areas as valid. Use multiple directives to specify additional ranges. Any code or data that falls outside the given range(s) will produce a warning (if the **-O** option is enabled) for each violating byte. Very useful for segmented memory devices, etc. Addr1 and addr2 may be specified in any order. The range defined will always be between the smaller and the higher values. The special keyword #OFF# removes all current definitions. See also **#VARIABLE**
#MESSAGE [*text*] **#HINT** [*text*]	Displays *text* on screen during the first pass of assembly when this directive is encountered in the source. Messages or hints are not written to the error file. They are meant to inform the user of options used or conditional paths taken. #HINT cannot be masked with the -Q+ option.
#NOWARN	Turns warnings off. Equivalent to the **-WRN-** command line option. See also **#WARN**
#OPTRELOFF	Disable «BRA/BSR instead of JMP/JSR» optimization warnings. Equivalent to the **-REL-** command line option.
#OPTRELON	Enable warning generation when an absolute branch or subroutine call (JMP or JSR) is encountered that could be successfully implemented using the relative form of the same instruction (BRA or BSR). This option is on by default. Equivalent to the **-REL+** command line option.
#OPTRTSOFF	Disable RTS-after-JSR/BSR optimization warning (default). Equivalent to the **-RTS-** command line option.
#OPTRTSON	Enable warning generation when a subroutine call (JSR or BSR) is immediately followed by a RTS. This option is off by default. Command-line option **-RTS+** does the same thing.

#PARMS [*char*|*SPACE*]	Allows changing the delimiter used to separate macro parameters when invoking the macro. If `char` is defined the new delimiter will be the same as `char`. If there is no character following the directive, the default parameter delimiter *(a comma)* will be used. To use a regular space as a parameter separator, the [*char*] part of the command should be the special keyword `SPACE` (case-insensitive). `#PUSH` and `#PULL` will save/restore the value of this setting.
#PPC	`#PPC` (stands for **P**reserve **PC**) simply keeps a copy of the current `:PC` value to be used later by the `:PPC` internal symbol. `#PUSH` and `#PULL` will save/restore the value of this setting.
#PROC **#ENDP**	Advances the `@@` local label counter. Nullifies the contents of the `~procname~` macro placeholder. See also **PROC** `#ENDP` closes the corresponding PROC section. See also **ENDP**
#PSP	`#PSP` (stands for **P**reserve **SP**) simply keeps a copy of the current `:SP` value to be used later by the `:PSP` internal symbol. *The `:PSP` symbol returns the difference between the then current `:SP` and the value saved with this directive. This can be used to de-allocate just the number of stack bytes that were pushed in between. This is only meant for use in `#SPAUTO` mode, which automatically adjusts the current value of the `:SP` internal symbol.* `#PUSH` and `#PULL` will save/restore the value of this setting.
#RENAME oldname,newname **#REMACRO** oldname,newname	Renames a macro from its current (old) name to a new name. An error message is issued if the old name is not a defined macro, the new name is a defined macro, or either name is an invalid symbol name. `#REMACRO` is the same as `#RENAME` except that it does NOT check if the new name exists. If it exists, there will now be one extra instance of that macro. *Note: The most recently defined macro of the same name is visible when more than one macro share the same name. `#DROP`-ping the macro always drops the visible instance, making a possible previous instance now visible.*

Tip: An example of where #RENAME might be useful:
Say, you have a library (or OS system) macro that is called many times in your application, but you want to modify that macro's behavior just for this one application. Your options are:
[1] Write a new (differently named) macro, and change all calls from the old macro to new macro. Problem: If some of these calls are inside shared library code, you can't change those calls, as it will affect other applications using those macros, as well. Too much work, and error prone.
[2] Alter the library macro to include the new behavior. Problem: Other applications may not like the new behavior.
[3] Use #RENAME in your application to have the old library macro change name just for this application's sake. Then, use the original name to write a brand new compatible macro but with the new behavior. It is also now possible for the new macro to 'borrow' the functionality of the old macro (by calling it internally as needed), so the new macro doesn't necessarily have to repeat the whole original macro body. This allows for an easy way to extend or replace any general-purpose library macros for each application, separately.

Example for #REMACRO that allows front-ending a previous macro to add code before and after the macro call.

```
a          macro
           #Message  Inside original ~0~
           endm

a          remacro
           #Message  Inside inner ~0~
           #rename   ~0~,_{:totalmacrocalls}_
           @@a
           #remacro  ~0~,~self~
           #Message  Inside inner ~0~
           endm

a          remacro
           #Message  Inside outer ~0~
           #rename   ~0~,_{:totalmacrocalls}_
           @@a
           #remacro  ~0~,~self~
           #Message  Inside outer ~0~
           endm

           @a
```

	`#Message ----------` `@a`
#S19FLUSH	Forces the immediate termination of an S-record line when encountered, rather than waiting for the record to reach the size specified by the **–R*n*** command line directive. This directive may be used to make identification of the end of code blocks easier when viewing the *.S19 file.
#S19RESET	Resets the S19 processor. Any used address ranges will be forgotten, and the same ranges used again. This directive may be used to combine multiple normally overlapping S19 files into one.
#S19WRITE [*text*]	Flushes the current S19 record, and then writes the text message into the S19 file on a line by itself. This directive may be used to add arbitrary text inside the S19 file, such as comments, special processing loader directives, etc.
#SIZE symbol[,expr]	Assigns the value of the expression to the size attribute of the specified previously-defined label. If no expression is present, then the difference between the current location and the label is used *(as if the expression was: *-LABEL)* which is the most common use of this directive. You can access the size attribute at a later time by using the internal symbol `::symbol` (where symbol is the symbol whose size you want to get.)
#SP [expr] **#SP1** [expr] **#SPAUTO** [expr][,expr] **#SPADD** [expr]	`#SP1` automatically adds one to all SP indexed offsets. It does this without affecting the current value of the `:SP` internal symbol. `#SP` without any expression cancels `#SP1` and `#SPAUTO` modes (reverts to default/normal operation). `#SP` followed by any expression (including a zero value) sets the `:SP` offset to the value of that expression but does not affect the current `#SPAUTO` mode. `#SPADD` adds a [signed] number to the current value of the `:SP` offset (regardless of mode). It does not reset the `:SPCHECK` variable, as with `#SPAUTO`. When `#SP1` is enabled, all SP indexed instructions use the same (zero-based) offsets as their corresponding X indexed instructions right after a `TSX` instruction. This allows using the same [named or numeric] offsets for both addressing modes to access the same

memory location(s)!

If the optional signed expression is present, its value will be added, also. This makes it easier to adjust for any stack depth changes, such as for subroutines or in-line stack changes.

`#SPAUTO` (or its shorter alias, `#SPA`) will automatically adjust the offset based on the instructions used. All push and pull instructions (including the extra ones) as well as all `AIS` instructions will automatically adjust the offset by as many bytes as required by each instruction. **Use the `#SP` directive (without <u>any</u> parameter offset, not even zero) to turn off the `#SPAUTO` mode and zero the SP offset *(or, use `#SPAUTO` with the special `#OFF#` parameter to turn off the `#SPAUTO` mode <u>without</u> changing the current SP offset.)***

`#SPAUTO` takes an optional second argument (any valid constant expression). If this value is specified, then the assembler (while in SPAUTO modes) will produce warnings when the stack depth increases beyond the value of the given expression. This value will remain active until it is explicitly turned off by using –1 in a subsequent `#SPAUTO` directive (e.g., `#SPAUTO ,-1`). The current value of this expression you can find in the internal variable `:SPLIMIT`
The maximum actual stack depth used will be in the `:SPMAX` variable which is reset only when `:SPLIMIT` is rewritten with a new value. This allows to check the maximum stack depth for a single routine, a collection of routines (e.g., in a module), or the whole application.

Manual alterations of the stack size, however (such as when you push an extra byte per loop iteration) cannot be automatically detected as the assembler will not follow your code's logic. In those cases, you'll have to adjust the offset 'manually' using `#SPADD` and an appropriate offset, like so:

```
#SPADD LOOPCOUNT-1
```

`#PUSH` and `#PULL` will save/restore the current setting of all modes

	of this option. **The assembler always starts in plain #SP mode (no offsets).** *See also internal symbols :SP and :SP1 and the simulated indexed modes ,ASP and ,LSP*
#SPCHECK	#SPCHECK checks the current value of the :SP internal symbol against the last used #SPAUTO value (found in :SPCHECK internal symbol), and issues a warning if the two numbers do NOT match, indicating a possible unbalanced stack situation (assuming correct placement of the relevant directives). The current difference between :SP and :SPCHECK is found in :SPFREE (e.g., use with AIS #:SPFREE) The warning also shows the number of bytes by which the stack is off. This can be used as a first-line of defense against unbalanced stack coding errors, especially in situations where there is heavy manipulation of the stack, and a visual inspection may prove confusing. Positive numbers indicate the stack contains so many extra bytes. Negative numbers indicate the stack is missing so many bytes. Hint: If you do not wish to use the #SPAUTO function for a particular section of code (or anywhere in your program) you can still temporarily place the #SPAUTO directive at the beginning of a code section to check, and the #SPCHECK at the end of the same code section, until you verify there are no related compilation warnings. Then you can remove the two directives (possibly even with the use of conditional directives), and continue with other coding work. *See also #SPAUTO*
#X [expr]	When #X is enabled (i.e., followed by a non-zero signed offset), all X indexed instructions will have that offset value automatically added to them (on top of whatever offset is actually specified with the instruction). This has a lot of potential uses, such as pointer adjustments (after TSX), or anytime the same constant needs to be

	added to a series of X-indexed instructions within a block of code. `#PUSH` and `#PULL` will save/restore the current setting of this option. **The assembler always starts in plain `#X` mode (no offsets).** *See also internal symbol `:X` and the simulated indexed mode `,AX`*
`#Y` `[expr]`	When `#Y` is enabled (i.e., followed by a non-zero signed offset), all Y indexed instructions will have that offset value automatically added to them (on top of whatever offset is actually specified with the instruction). This has a lot of potential uses, such as pointer adjustments (after TSY), or anytime the same constant needs to be added to a series of Y-indexed instructions within a block of code. `#PUSH` and `#PULL` will save/restore the current setting of this option. **The assembler always starts in plain `#Y` mode (no offsets).** *See also internal symbol `:Y` and the simulated indexed mode `,AY`*

#PAGE	Outputs a Form Feed (ASCII 12) character followed by a Carriage Return (ASCII 13) in the *.LST file just before displaying the line that contains this directive.
#PUSH	Pushes on an internal stack the **current segment** and the current settings of the following directives: **MAPx**, **LISTx**, **CASEx**, **EXTRAx**, **SPACESx**, **OPTRELx**, **OPTRTSx**, **[NO]WARN, TRACEx**, **MACRO**, **@MACRO**, **MCF**, **MACROSHOW, MACROHIDE, :PSP, :PPC, TRACE[ON/OFF], MLISTx**, and **TABSIZE**. Useful in included files that want to change any of these options without affecting parent/sibling files. See also **#PULL**
#PULL	Pulls from an internal stack the most recently pushed options. See also **#PUSH**
#RAM	Activation of the RAM segment. Default starting value is $0000.
#ROM	Activation of the ROM segment. Default starting value is $D000. This is the default segment if none is specified.
#SEG*n*	Activation of the SEGn segment (n is a number from 0 through 9). Default starting value for all ten segments is $0000.
#TABSIZE *n*	Specifies the field width of tab stops used in the source file. Proper use of this directive ensures that the *.LST files generated by ASM11 are formatted in the same way as your source files appear in your text editor. This directive overrides the setting of the **–T*n*** command line option at the point in the source file(s) in which it is encountered.
#TEMP [expr] **#TEMP1** [expr] **#TEMP2** [expr]	#TEMP simply assigns any value (possibly the result of an non-forward expression) to the internal general-purpose :TEMP variable (similarly, for #TEMP1 and #TEMP2). If no expression follows #TEMP, :TEMP is zeroed. :TEMP can be used any time in lieu of defining any 'helper' symbol for intermediate calculations (either inside or outside macros). The only restriction is that :TEMP always refers to the most recent #TEMP directive, so it cannot be used to look forward. Although :TEMP is a single variable, its use is transparent in relation to macros. In other words, changing :TEMP from within any macro does not affect the value of :TEMP outside all macros, or macros above the current level.

	Although macros inherit their initial value of `:TEMP` from their higher level (either a caller macro, or normal code), they do not affect their parent's `:TEMP` value, so you can use it without worrying about side effects from any intermediate macro calls. `:TEMP` is also assigned *indirectly* when used as label with any of the following directives/pseudo-ops: `NEXT`, `NEXP`, `SETN`, and `#AIS`
#TRACEON **#TRACEOFF**	**#TRACEON** enables generation of source-line information in the *.MAP file for any code found in the body of macros following this directive. The map info is generated in such a way that while tracing the debugger will display the actual source of the macro. This can be used globally (to affect all macro invocations), inside a specific macro (to debug that one macro), or around a specific macro invocation (to debug that one macro call.) **#TRACEOFF** turns this option off making macros appear as a single line in the debugger. **#TRACEOFF** is the default state when assembly is started.
#VARIABLE `addr1 [addr2]` **#VARIABLE** `#OFF#`	Maps a location (or range, if addr2 is also supplied) of variable allocation area (normally in RAM) as valid. Use multiple directives to specify additional ranges. Any `RMB` or `DS` definitions that fall *(fully or partially)* outside the given range(s) will produce a warning (if the **-o** option is enabled) for each such definition. Addr1 and addr2 may be specified in any order. The range defined will always be between the smaller and the higher values. The special keyword `#OFF#` removes all current definitions. See also **#MEMORY**
#VECTORS	Activation of the VECTORS segment. Default starting value is $FFD6.
#WARN	Turns warnings on. Equivalent to the **-WRN+** command line option. See also **#NOWARN**
#WARNING [*text*]	Similar to the **#ERROR** directive, but generates an assembler warning message instead of an error message.
#XRAM	Activation of the XRAM segment. Default starting value is $2000.
#XROM	Activation of the XROM segment. Default starting value is $8000.
#[!]UNDEF `symbol[,symbol]*`	Undefines one or more symbols. If a symbol is not currently defined, a warning will be issued (to protect from possible typing errors) unless the `!UNDEF` form is used. Careless, or simply wrong use of this directive can lead to multiple

	side errors or warnings (please note this is a two-pass assembler). If you simply want to redefine the value of a symbol, prefer using the `SET` pseudo-op, rather than `#UNDEF` followed by another symbol definition. `#UNDEF` can be used, for example, to completely remove unrelated or conflicting conditionals.

Note: [text] in directives and all strings may contain *nested* expressions enclosed in curly brackets, e.g. *{expr}*. The expression may not contain spaces (regardless of the `-SP` option state, or `#SPACESON` directive. An optional format modifier (case-insensitive) within parentheses after the expression can force the display in the specified format. **(D)** for default/decimal, **(H)** for hex, **(S)** for signed decimal, **(1)** thru **(4)** (or, thru **(9)** for the 32-bit versions) for the corresponding number of decimal places after division by 10^n where *n* is a number from 1 to 4 (or 9), **(X)** for expanded, **(Fn)** for space left filled, and **(Zn)** for zero left filled, where n is optional (default is 2) and can range from 1 to 0 (0 meaning 10). Some examples using this feature:

```
ROM EQU $F000
#Message ROM is at {ROM}
```
will display:
```
ROM is at 61440
```

Adding a format modifier will have the following effect:

```
#Message ROM is at {ROM(x)}
```
will display:
```
ROM is at 61440 [$F000]
```

```
#Message ROM is at {ROM(d)}
```
will display:
```
ROM is at 61440
```

```
#Message ROM is at {ROM(h)}
```
will display:
```
ROM is at $F000
```

```
#Message ROM is at {ROM(s)}
```
will display:
```
ROM is at -4096
```

```
#Message Clock: {:year}-{:month(z)}-{:date(z)} {:hour(z)}:{:min(z)}:{:sec(z)}
```
will display something like:
```
Clock: 2013-11-05 13:00:00
```

It can also be used in strings, like so:

```
VERSION equ 101 ;Firmware version as x.xx
MsgVersion fcs 'Firmware v{VERSION(2)}',LF
```
is equivalent to

MsgVersion fcs 'Firmware v1.01',LF

but it will automatically adjust the `MsgVersion` string each time the symbol `VERSION` changes value. No need to re-adjust all relevant messages manually.

An expression that cannot be evaluated (due to forward references or undefined symbols) will display as three question marks (**???**) in directives, but no error or warning message will be issued. When used in strings, however, errors will be displayed as usual.

To prevent an expression evaluation in directives, enclose the [text] that contains the curly brackets within quotes.

To prevent an expression evaluation in strings, break the string into two so that both curly brackets are not part of the same string, e.g.:

instead of **fcc '{Hello}'** which tries to evaluate the symbol *Hello* use: **fcc '{','Hello}'**.

Internally defined symbols

Some special internal symbols are defined by the assembler. All such symbols begin with a colon (:) character. Currently, the following internal symbols are defined:

- `::` (without a symbol following) returns the current (dynamically assigned) stack offset. Very useful mostly in `#SPAUTO` mode so that you can assign labels to stack contents as they are created. (Same as `1-:SP` in `#SP[AUTO]` modes, or `0-:SP` if in `#SP1` [sub-]mode.) *Note: If any push instruction is followed by a label, that label will be `SET` to the current `::` value (must be in `#ExtraOn` mode).*
- **`::symbol`** (where symbol is any previously defined symbol) returns the current 'size' for the given symbol. A symbol's size is determined either automatically (e.g., `RMB` pseudo-instructions), or manually via the `#SIZE` directive.
- **`:SP`** returns the current offset of the `#SP` or `#SP1` directives. *This value is the basis for several other internal symbols.*
- **`:SPLIMIT`** returns the currently effective value of the `#SPAUTO` stack depth check option (i.e., the optional 2nd parameter of the `#SPAUTO` directive.) `#PUSH`/`#PULL` save/restore this value.
- **`:SPMAX`** returns the maximum used stack depth since the last time `:SPLIMIT` was explicitly set (even if to the same value it had already.) You can use this internal variable to find the maximum stack depth for a single routine, a collection of routines (e.g., a whole module), or even your whole application's. *Keep in mind, however, that it only counts stack depth in a linear fashion, i.e., without considering possible subroutine calls, recursion, or other indirect methods of altering the stack, such as the `LDHX #STACKTOP / TXS` sequence.*
- **`:SPX`** returns `:SP-1` when in `#SP[AUTO]` modes and `:SP-0` when in `#SP1` [sub-]mode. Useful with `#X` as in `#X :SPX`. *Alternatively (and preferably), you may use the `,SPX` simulated indexed mode, which does not depend on the `:SPX` value, and which is actually `X`-indexed mode but stack-relative to the most recent `TSX` instruction, and possible subsequent `INX/GETX/GIVE` instructions. (Note: there are many ways to alter the contents of the X index register; the assembler cannot automatically account for all those possibilities; use `#X expr` where needed to manually adjust the offset, and use the plain `X`-indexed mode). This feature provides a very simple way of creating `SP` relative instructions to `,X` relative (by simply using `,SPX` and TSX anywhere before these instructions. All offsets are automatically adjusted.)*
- **`:SPY`** returns `:SP-1` when in `#SP[AUTO]` modes and `:SP-0` when in `#SP1` [sub-]mode. Useful with `#Y` as in `#Y :SPY`. *Alternatively (and preferably), you may use the `,SPY` simulated indexed mode, which does not depend on the `:SPY` value, and which is actually `Y`-indexed mode but stack-relative to the most recent `TSY` instruction, and possible subsequent `INY/GETY/GIVEY` instructions. (Note: there are many ways to alter the contents of the Y index register; the assembler cannot automatically account for all those possibilities; use `#Y expr` where needed to manually*

adjust the offset, and use the plain Y–indexed mode). This feature provides a very simple way of creating `SP` relative instructions to `,Y` relative (by simply using `,SPY` and TSY anywhere before these instructions. All offsets are automatically adjusted.)

- ***`:TSX`** is similar to `:SPX` but, although relative to the most recent `TSX` instruction, and possible subsequent `INX/GETX` instructions (like `:SPX`), it disregards possible following stack depth changes, unlike `:SPX`. (Note: there are many ways to alter the contents of the X index register; the assembler cannot automatically account for all those possibilities; use `#X expr` where needed to manually adjust the offset, and use the plain `X`–indexed mode).*
- ***`:TSY`** is similar to `:SPY` but, although relative to the most recent `TSY` instruction, and possible subsequent `INY/GETY` instructions (like `:SPX`), it disregards possible following stack depth changes, unlike `:SPY`. (Note: there are many ways to alter the contents of the Y index register; the assembler cannot automatically account for all those possibilities; use `#Y expr` where needed to manually adjust the offset, and use the plain `Y`–indexed mode).*
- **`:SPCHECK`** returns the actual offset used with the most recent `#SPAUTO` directive.
- **`:SPFREE`** returns the current stack depth change (same as `:SP-:SPCHECK`). For example, you may use it with the `AIS` instruction to free so many bytes of stack. (The symbol `:SP` alone will not work for this purpose – releasing remaining stack bytes – unless `#SPAUTO` is used with a zero offset, while `:SPFREE` works, regardless of the initial offset.)
- **`:AIS`** returns the current stack depth change since the last `#AIS` directive *(when given without any parameters)*. For example, you may use it with a new [normally, stack-reducing] `GIVEX/GIVEY` instruction to free so many bytes of stack. *`:AIS` is updated automatically after each #AIS directive, losing whatever previous value was in `:AIS`, and it can be used to free whatever stack bytes remain since the last #`AIS` directive (used like so:, `GIVEX #:AIS`). `#SP`, `#SPAUTO`, and `#SP1` reset the `:AIS` symbol to zero or the value of the parameter used with the corresponding directive until the next #`AIS` directive.*
- **`:PSP`** returns the current stack depth change since the last `#PSP` directive. For example, you may use it with the `AIS` instruction to free so many bytes of stack. *Unlike `:SPFREE` which is related to the automatically updated `:SPCHECK` during any `#SPAUTO` directive, `:PSP` is only updated manually with the `#PSP` directive, and can be used locally (eg., around a sub-routine call) to free the number of stack bytes for only a specific section of code (eg., whatever parameters were pushed on the stack for use by the sub-routine).*
- **`:SP1`** returns the current offset of the `#SP` or `#SP1` directives (like `:SP`), *but also adds one only if we're currently in the `#SP1` mode.* This value is always the true effective offset for both `#SP` and `#SP1` modes.
- **`:X`** returns the current offset of the `#X` directive.
- **`:Y`** returns the current offset of the `#Y` directive.
- **`:YEAR`** returns the year at assembly time (e.g., 2019) *Hint: Use `:YEAR\100` for two-digit year.*
- **`:MONTH`** returns the month at assembly time (e.g., 5)

- **:DATE** returns the date at assembly time (e.g., 29)
- **:HOUR** returns the hour at assembly time (e.g., 13)
- **:MIN** returns the minute at assembly time (e.g., 0)
- **:SEC** returns the second at assembly time (e.g., 0)
- **:CPU** returns a number representing the CPU type (6811)
- **:CRC** returns the current value of the running user CRC
- **:S19CRC** returns the current value of the running S19 CRC.
- **:CYCLES** returns the current value of the cycles counter, and then it is reset to zero.
- **:OCYCLES** returns the older value of the cycles counter (but does not reset it).
- **:TOTALMACROCALLS** returns the current value of the total macro invocations. Use it for display, or even to restrict macro use (e.g., `#IFNZ :TOTALMACROCALLS … #ERROR No macros allowed for this application … #ENDIF`).
- **:MACRONEST** returns the current value of the macro (chain) 'loop level' regardless if calling the same, or a different macro (think of it as the 'nesting level'). A value of zero is returned if used outside any macros. First level is number 1. Each time the top-level macro is called, the number is reset to 1. Each time the same or a different macro is called from within the current macro, the number is incremented by 1. The macro (chain) can also initialize itself during, say, count one.
- **:MACROLOOP** (or, simply, **:LOOP**) is similar to :MACRONEST but it returns the current value of the macro 'loop level' only for the current macro. A value of zero is returned if used outside any macros. First level is number 1. Each time the macro is called from outside any macros, or from a different macro, the number is reset to 1. Each time the macro calls itself (by either a chained macro call, or the MTOP directive), the number is incremented by 1. This can be used as an automatic loop counter. The macro can also initialize itself during, say, count one. This differs from :MACRONEST in that chained macro calls will restart this counter for each new macro. This counter is also reset with a %macro syntax call.
- **:MLOOP** is similar to :LOOP but it is only affected by the MDO and MLOOP keywords. First count is number 1. Each time the MDO keyword is encountered, the number is reset to 1. Each time the MLOOP keyword is encountered, the number is incremented by 1. This can be used as an automatic loop counter. This counter is also reset with a %macro syntax call.
- **:MEXIT** holds the most recent MEXIT defined value. :MEXIT is reset to zero each time a macro is (re)entered, but its value can be changed by MEXIT instructions that specify an explicit expression. This feature can be used to pass back to the higher level any value from inside a (nested) macro (such as success/error status, the result of some computation, etc.) without using any label definitions.
- **:MACROINDEX** (or **:MINDEX**) returns the current value of the current macro's number of invocations. A value of zero is returned if used outside any macros. First call of each macro is number 1. If the specific macro is dropped and re-created, the number is reset (it is, afterall, a new macro). An example use is to create different labels at each invocation (not to be confused with

automatic $$$ label generation, which assumes values based on :TOTALMACROCALLS and cannot be guaranteed to take sequential values between consecutive calls of the exact same macro since other macros may have increased the counter in between), or instruction offsets (e.g., with the special ad-hoc macro named "?"), etc. This counter is also reset with a %macro syntax call.

- **:INDEX** returns the next value of the current macro's internal user index. A value of zero is returned if used outside any macros. First use in each macro is number 1. If the specific macro is dropped and re-created, the number is reset (it is, afterall, a new macro). Its use is similar to :MACROINDEX but there is a significant difference. :INDEX is only updated each time it is accessed, regardless of how many times the macro is actually called. So, if used inside a conditional block of code, it will only be incremented when that part is expanded. *Note: Because of the auto-increment on access, if you want to use the same value more than once in the same macro invocation, you must first assign the value to some label, and then use the label, instead.* This counter is also reset with a %macro syntax call.
- **:0** to **:9** return the length of the text of the corresponding macro parameter. You can use it alone or along with the ~n.s.l~ parameter placeholder. This can only be used from within a macro. *It is not recognized as valid symbol outside a macro.*
- **:DOW** returns the day-of-week number at assembly time, from zero (Sunday) to six (Saturday).
- **:PC** returns the current program counter (same as *) but can be used even in expressions where the use of * is ambiguous.
- **:PPC** returns the previously saved program counter (see the #PPC directive). It can be used to get the byte distance between any two points without having to define a symbol just for this. It is also useful inside frequently called macros; for example, to avoid the use of a macro local label definition for simple loops (helps keep the symbol table smaller in large applications).
- **:PROC** returns the current value of the internal local symbol counter (See PROC and #PROC).
- **:MAXPROC** returns the currently maximum value of the internal local symbol counter (See PROC and #PROC).
- **:LABEL** returns the length of the ~label~ placeholder's content used inside macros.
- **:TEMP :TEMP1 :TEMP2** return the current value of the corresponding internal user-defined assembly-time variable. (See the #TEMP directive for more details.)
- **:TEXT** returns the length of the current text of the ~text~ macro parameter (only from within macros.)
- **:LINENO** returns the current file line number.
- **:MAXLABEL** returns the current value of the maximum label length.
- **:MLINENO** returns the current macro line number (only from within macros).
- **:N** returns the current macro number of contiguous arguments (only from within macros).
- **:NN** returns the current macro number of all arguments (only from within macros).
- **:ANRTS** returns the address of the most recent RTS instruction (i.e., always points back).

- **`:ROM, :RAM, etc.`** All segment directives have a corresponding internal variable that returns the current value of that segment.
- **`:OFFSET`** returns the current S19 addressing offset from the physical address (see `ORG`).
- **`:WIDTH`** returns the current width of the console screen. Useful for formatting user messages.

Notes about `:cycles`:

- The cycles counter is reset to zero right after it is accessed. To count cycles for a section of code, you must access `:cycles` twice, once before the code section to reset its value to zero (if not already zero from a previous access to `:cycles` or a `#CYCLES` directive), and once right after the code section to get the accumulated cycles.
- Because of the auto-reset on access, if you need to use the same value in more than one place at a time (e.g., code and `#MESSAGE` directive), you must assign it to a label first, then use the label.
- The obvious advantage is that if you alter code as in the example loop below (e.g., by adding conditional early escape code inside the loop), it will still be timed correctly without requiring a manual adjustment of the delay constant. Another advantage is that conditionally enabled code will be accounted for correctly in all cases, again without requiring a manual recalculation for each conditional case.
- Example use of `:cycles` that automatically calculates the appropriate delay constant:

```
                #Cycles                         ; reset cycles counter
Delay10ms       pshx
                ldx     #10*BUS_KHZ-?ExtraCycles/?LoopCycles
?ExtraCycles    equ     :cycles                 ; grab counter (and reset)
?Delay.Loop     dex
                bne     ?Delay.Loop
?LoopCycles     equ     :cycles                 ; grab counter (and reset)
                pulx
                rts
?ExtraCycles    set     ?ExtraCycles+:cycles
```

(SET instead of EQU allows re-using symbols, so you can use it to accumulate related cycles.)

Example assembly code for calculating user CRC

```
;*******************************************************************************
; Purpose: Calculate the same user CRC as that produced by ASM11
; Input  : X -> First byte of block
;        : Y -> Last byte of block
;        : D = Initial/Previous CRC
; Output : D = updated CRC
; Note(s): Call repeatedly for different address ranges, if skipping sections
; Call   :          ldd       #CRC
;        :          ldx       #StartAddress
;        :          ldy       #EndAddress
;        :          jsr       GetAsmCRC

?                   set       0
?StartAddress       next      ?,2
?EndAddress         next      ?,2
?CRC                next      ?,2

GetAsmCRC           pshx
                    pshy

                    pshd                          ;CRC
                    pshy                          ;ending address
                    pshx                          ;starting address
                    tsy                           ;Y -> stack frame

?GetAsmCRC.Loop     cmpx      ?EndAddress,y
                    bhi       ?GetAsmCRC.Exit

                    sta       COPRST              ;in case of many iterations

                    lda       ,x
                    beq       ?GetAsmCRC.Next
                    cmpa      #$FF
                    beq       ?GetAsmCRC.Next

                    ldb       ?StartAddress+1,y
                    mul                           ;low address with data byte
                    addd      ?CRC,y
                    std       ?CRC,y

                    lda       ,x
                    ldb       ?StartAddress,y
                    mul                           ;high address with data byte
                    addb      ?CRC,y
                    stb       ?CRC,y

?GetAsmCRC.Next     inx
                    stx       ?StartAddress,y
                    bra       ?GetAsmCRC.Loop

?GetAsmCRC.Exit     pulx
                    puly
                    puld

                    puly
                    pulx
                    rts
```

Example coding for skipping CRC calculation for volatile sections

```
?crc                set       :crc                ;use SET, not EQU

       ;CODE/DATA TO SKIP FROM CRC CALCULATION HERE

                    #CRC      ?crc
```

Expression Operators and Other Special Characters Recognized by Asm11

- Expressions are evaluated in the order they are written (left to right).
  All operators have equal precedence.
- Avoid inserting spaces between values and operators (unless using -SP+ switch and semicolon beginning comments).

Operator	Description
+	Addition
-	Subtraction When used as a unary operator, the 2's complement of the value to the right is returned.
*	Multiplication Can also be used to represent the current location counter.
/	Integer Division (ignores remainder)
\	Modulus (remainder of integer division)
=	'Equal to' comparison for the `$IF` directive.
<>	'Not equal to' comparison for the `$IF` directive.
>=	'Greater than or equal to' comparison for the `$IF` directive.
>	Shift right – operand to the left is shifted right by the count to the right. Also used to specify extended addressing mode. 'Greater than' comparison for the `$IF` directive.
<=	'Less than or equal to' comparison for the `$IF` directive.
<	Shift left – operand to the left is shifted left by the count to the right. Also used to specify direct addressing mode. 'Less than' comparison for the `$IF` directive.
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR (exclusive OR)
~	Swap high and low bytes (unary): `~$1234 = $3412` Useful for converting word values from big-endian to little-endian or the inverse.
[[	Extract low 16 bits (unary): `[[$123456 = $3456`
]]	Extract high 16 bits (unary): `]]$123456 = $0012`
[	Extract low 8 bits (unary): `[$1234 = $34`
]	Extract high 8 bits (unary): `]$1234 = $12`

$	Interpret numeric constant that follows as a hexadecimal number. Can also be used to represent the current location counter.
%	Interpret numeric constant that follows as a binary number
' ` "	Any one of these characters (single, back, or double-quote) may be used to enclose a string or character entity. The character used at the start of the string must be used to end it.
#	Specifies immediate addressing mode
@	Specifies direct addressing mode (same as «**<**»)

ASM11 Extended Instruction Set

The instructions listed below are not actually new instructions, rather, internal macros that generate one or more 68HC11 CPU instructions. These instructions are only recognized if the extended instruction set option is enab*l*ed (**-X+** command line option or **#EXTRAON** processing directive) , and are used just like normal instructions, but NOT like user-defined macros.

Mnemonic/Syntax	Description	
AIX *#word*	Add Immediate X the 16-bit value #word (signed or unsigned). Equivalent to **XGDX** / **ADDD** *#word* / **XGDX**	
AIY *#word*	Add Immediate Y the 16-bit value #word (signed or unsigned). Equivalent to **XGDY** / **ADDD** *#word* / **XGDY**	
LDA *operand*	Same as:	**LDAA** *operand*
LDB *operand*	Same as:	**LDAB** *operand*
STA *operand*	Same as:	**STAA** *operand*
STB *operand*	Same as:	**STAB** *operand*
ORA *operand*	Same as:	**ORAA** *operand*
ORB *operand*	Same as:	**ORAB** *operand*
PSHD	Push D:	**PSHB** / **PSHA**
PULD	Pull D:	**PULA** / **PULB**
CMPD *operand*	Same as:	**CPD** *operand*
CMPX *operand*	Same as:	**CPX** *operand*
CMPY *operand*	Same as:	**CPY** *operand*
CLRD	Clear D:	**CLRA** / **CLRB**
CLRX	Clear X:	**LDX #0**
CLRY	Clear Y:	**LDY #0**
COMD	1's Complement D:	**COMA** / **COMB**
NEGD	2's Complement D:	**COMA** / **COMB** / **ADDD #1**
XGAB	Exchange A and B:	**PSHA** / **TBA** / **PULB**
ROLD	Rotate Left D:	**ROLB** / **ROLA**
RORD	Rotate Right D:	**RORA** / **RORB**
INCD	Increment D:	**ADDD #1**
INCX	Increment X:	**INX**
INCY	Increment Y:	**INY**
DECD	Decrement D:	**SUBD #1**

DECX	Decrement X: **DEX**
DECY	Decrement Y: **DEY**
LBRA *addr16*	Long relative branch: *(22 bytes/69 cycles)* **Warning!** Generates considerable code, use with care
LBSR *addr16*	Long relative subroutine call: *(32 bytes/92 cycles)* **Warning!** Generates considerable code, use with care
GETX *#word*	Get #word bytes of stack storage pointed to by X for temporary use. Equivalent to **TSX** / **XGDX** / **SUBD #word** / **XGDX** / **TXS**
GETY *#word*	Get #word bytes of stack storage pointed to by Y for temporary use. Equivalent to **TSY** / **XGDY** / **SUBD #word** / **XGDY** / **TYS**
GIVEX *#word*	Give (back) #word bytes of stack storage pointed to by X. Equivalent to **TSX** / **XGDX** / **ADDD #word** / **XGDX** / **TXS**
GIVEY *#word*	Give (back) #word bytes of stack storage pointed to by Y. Equivalent to **TSY** / **XGDY** / **ADDD #word** / **XGDY** / **TYS**
JCC *addr16*	Jump equivalent to **BCC** (BCS $+5 followed by JMP *addr16*)
JCS *addr16*	Jump equivalent to **BCS** (BCC $+5 followed by JMP *addr16*)
JEQ *addr16*	Jump equivalent to **BEQ** (BNE $+5 followed by JMP *addr16*)
JGE *addr16*	Jump equivalent to **BGE** (BLT $+5 followed by JMP *addr16*)
JGT *addr16*	Jump equivalent to **BGT** (BLE $+5 followed by JMP *addr16*)
JHI *addr16*	Jump equivalent to **BHI** (BLS $+5 followed by JMP *addr16*)
JHS *addr16*	Jump equivalent to **BHS** (BLO $+5 followed by JMP *addr16*)
JLE *addr16*	Jump equivalent to **BLE** (BGT $+5 followed by JMP *addr16*)
JLO *addr16*	Jump equivalent to **BLO** (BHS $+5 followed by JMP *addr16*)
JLS *addr16*	Jump equivalent to **BLS** (BHI $+5 followed by JMP *addr16*)
JLT *addr16*	Jump equivalent to **BLT** (BGE $+5 followed by JMP *addr16*)
JMI *addr16*	Jump equivalent to **BMI** (BPL $+5 followed by JMP *addr16*)
JNE *addr16*	Jump equivalent to **BNE** (BEQ $+5 followed by JMP *addr16*)
JPL *addr16*	Jump equivalent to **BPL** (BMI $+5 followed by JMP *addr16*)
JVC *addr16*	Jump equivalent to **BVC** (BVS $+5 followed by JMP *addr16*)
JVS *addr16*	Jump equivalent to **BVS** (BVC $+5 followed by JMP *addr16*)
CLS	Clear S flag: **PSHA** / **TPA** / **ANDA #$7F** / **TAP** / **PULA**
CLX	Clear X flag: **PSHA** / **TPA** / **ANDA #$BF** / **TAP** / **PULA**
PULL	Same as: **PULD** / **PULX** / **PULY**
PUSH	Same as: **PSHY** / **PSHX** / **PSHD**
SES	Set S flag: **PSHA** / **TPA** / **ORAA #$80** / **TAP** / **PULA**
SEX	Sign extend B to A: **CLRA** / **TSTB** / **BPL ?** / **COMA** / **?**

WAIT	Enter WAIT mode:	**CLI** / **WAI**
OS *byteval*	Operating system call:	**SWI** / **DB** *byteval*
OS11 *wordval*	Operating system call:	**SWI** / **DW** *byteval*
OSW *wordval*	Operating system call:	**SWI** / **DW** *wordval*
XGXY	Exchange X and Y:	**XGDX** / **XGDY** / **XGDX**

ASM11-generated Error and Warning Messages

This section provides the lists of error and warning messages.

Errors inform the user about problems that prevent the assembler from producing usable code. If there is even a single error during assembly, no files will be created (except for the ERR file, if one was requested).

Warnings inform the user about problems that do not prevent the assembler from producing usable code but the code produced may not be what was intended, or it may be inefficient. A program that has warnings may be totally correct and run as expected.

Errors and warnings that begin with '`USER:`' are generated by `#ERROR` and `#WARNING` directives, respectively. The source code author decides their meaning and importance.

In the lists below, what's enclosed in angle brackets (< and >) is a 'variable' part of the message. That is, it is different depending on the source line to which the error or warning refers.

The order the messages appear below is random. Some messages have similar meanings; they simply result from different checks of the assembler.

E R R O R S

1. ***Invalid binary number***

The string following the % sign is not made up of zeros and/or ones.

2. ***Binary number is longer than 16 bits***

A binary number may have no more than sixteen significant digits. Leading zeros are ignored.

3. ***"<SYMBOL>" not yet defined, forward refs not allowed***

`RMB` and `DS` directives may not refer to forward defined symbols. You must define the symbol(s) used in advance.

4. ***Bad <MODE> instruction/operand "<OPCODE> <OPERAND>***

The instruction and operand addressing mode combination is not a valid one, or you have turned the -X option (EXTRAx directive) off. For example, `TST #4` will show «*Bad IMMEDIATE instruction/operand "`TST 4`*"» because although `TST` is a valid instruction, it does not have an immediate addressing mode option.

5. ***Could not close MAP file***

For some reason, the MAP file could not be closed. Possibly some disk problems (check available space, etc.) If you can't figure out what's wrong and still must assemble, turn off the MAP with the `-M-` option.

6. ***Could not close SYM file***

For some reason, the SYM file could not be closed. Possibly some disk problems (check available space, etc.) If you can't figure out what's wrong and still must assemble, turn off the SYM with the `-S-` option.

7. ***Could not create MAP file <FILEPATH>***

For some reason, the MAP file could not be created. Possibly some disk problems (check available space, etc.) If a MAP file of the same name already exists it probably has a read-only attribute or is somehow locked by the system.

8. ***Could not create SYM file <FILEPATH>***

For some reason, the SYM file could not be created. Possibly some disk problems (check available space, etc.) If a SYM file of the same name already exists it probably has a read-only attribute or is somehow locked by the system.

9. ***Expression error***

Something is wrong with the attempted expression, or an expression is altogether missing.

10. ***Invalid argument for DB directive***

The value or expression supplied is not correct.

11. ***Invalid argument for EQU/EXP directive***

The value or expression supplied is not correct.

12. ***Invalid first argument***

The first value or expression supplied is not correct.

13. ***Invalid second argument***

The second value or expression supplied is not correct.

14. ***Missing value between commas***

Two or more commas without a value in between.

15. ***Possibly duplicate symbol "<SYMBOL>"***

The symbol shown has already been defined. The word 'possibly' suggests that a symbol may have been truncated to 19 characters, and thus not appear duplicate to the user, only to the assembler. It also suggests that the original may have been written for case-sensitive assembly but you turned the option off.

16. ***Repeater value is invalid***

The repeater value (the `:n` part of the opcode) is a positive integer number.

17. ***Symbol "<SYMBOL>" contains invalid character(s)***

The symbol shown contains characters that are used in special ways and, therefore, cannot be part of a symbol because they will cause ambiguities. For example, a quote within a symbol is not allowed.

18. ***Undefined symbol "<SYMBOL>" or bad number***

The string shown is either a symbol that hasn't been defined at all, or it is a number that has some error, for example: `$ABCH` and `$FFFFF` are not valid hex numbers. The first contains an invalid character while the second is greater than 16 significant bits ($FFFF).

19. ***USER: <USER TEXT>***

This is a user generated error via the `#ERROR` directive.

20. ***Comma not expected***

A comma was found in an unexpected position within the operand. Possibly using more arguments than required.

21. ***Syntax error***

Some symbol is confusing the assembler. For example, `FCB #$FF` will give a syntax error because the # indicates immediate addressing mode which makes no sense for an FCB directive (the correct is `FCB $FF`).

22. ***Empty string not allowed***

An empty string (two quotes next to each other) is not allowed because there is no value that can be generated from it.

23. ***Could not open include file <FILEPATH>***

The [path and] file shown could not be located or opened. If the file exists, it may be locked by some other program (under Windows, the file could be loaded in an editor).

24. ***ELSE without previous Ifxxx***

An `#ELSE` directive was encountered that does not match any unmatched `#IF` directive.

25. ***ENDIF without previous Ifxxx***

An `#ENDIF` directive was encountered that does not match any unmatched `#IF` directive.

26. ***Forward references not allowed***

The `#MEMORY/#VARIABLE` and other directives do not accept forward references.

27. ***Incomplete argument for Bit Instruction (commas?)***

`BSET`, `BCLR`, `BRSET`, and `BRCLR` require commas between each part of the operand. You have either left the commas out or forgotten to supply all the parts of the operand. Assembling code written for Mot's AS11 will produce a lot of these.

28. ***Invalid expression(s) and/or comparator***

The expression or comparator used in the `#IF` directive is incorrect.

29. ***Missing branch address***

`BRSET` and `BRCLR` require a target address for branching to but one was not supplied. The branch target is the last part of the operand and it can be any valid expression.

30. ***Missing INCLUDE filename***

An `#INCLUDE` directive was supplied without any [path and] filename.

31. ***Missing required first address***

A `#MEMORY/#VARIABLE` directive was encountered without any value or expression.

32. ***Repeater value out of range (1-32767)***

The repeater value (the :n part of the opcode) must be from 1 to 32767.

33. ***Required string delimiter not found***

You have supplied only one quote to a string, or the string is inappropriately separated from the previous or next operand. For example: `'ABC'CR` lacks a comma between the quote and the CR symbol. So, the found string `ACB'CR` is invalid.

34. ***Symbol "<SYMBOL>" does not start with A..Z, . or _***

All symbols must start with one of the above characters. (Local symbols start with a ?)

35. ***Symbol "<SYMBOL>" is reserved for indexing modes***

You have used a symbol named X or Y. These names are not allowed because they cause ambiguities with the X and Y registers in the various indexed mode instructions.

36. ***Too many include files. Maximum allowed is 99***

The maximum number of `INCLUDE` files is 99 (regardless of nesting level). You have gone over this number. Possible solution: Combine related files together as required. Keep in mind that although the assembler allows this many files to be included, a lot of programs cannot handle these many files in the MAP files.

37. ***Division by zero***

The expression used contains a division by zero after the / operator.

38. ***MOD division by zero***

The expression used contains a division by zero after the \ operator.

39. ***"<SYMBOL>" is too far back [<VALUE>], use jumps***

The target of a branch instruction is too far back by as many bytes as shown. Either get it closer to the target, or use a jump instead. Some instructions, like `BRCLR` or `BRSET` do not have an equivalent jump so you must use an intermediate 'jump hook' instead.

40. "<SYMBOL>" is too far forward [<VALUE>], use jumps

The target of a branch instruction is too far forward by as many bytes as shown. Either get it closer to the target, or use a jump instead. Some instructions, like `BRCLR` or `BRSET` do not have an equivalent jump so you must use an intermediate 'jump hook' instead, i.e., branch to a nearby JMP instruction that jumps to the desired destination.

41. Invalid argument for DW or FDB directive

The value or expression supplied is not correct.

42. Invalid argument for ORG directive

The value or expression supplied is not correct.

43. Invalid argument for RMB or DS directive

The value or expression supplied is not correct.

44. Invalid argument for END directive

The value or expression supplied is not correct.

W A R N I N G S

48. Direct mode wasn't used (forward reference?)

Automatic Direct Mode detection requires that any symbol(s) used be defined in advance. You should either define the referenced symbol(s) earlier in your code, or use the Direct Mode Override (<) to force the assembler to use Direct Addressing Mode.

49. Label on left side of END line ignored

The `END` directive does not take a label. If one is used it will be ignored (it will not be defined).

50. Label on left side of ORG line ignored

The `ORG` directive does not take a label. If one is used it will be ignored (it will not be defined).

51. Trailing comma ignored

A multiple-parameter pseudo-instruction was used (such as `FCB` and `DW`) and a comma was found at the end of the parameter list. This may indicate both commas and spaces separate the list. You must either remove the spaces or assemble with `#SPACESON` (or the `-SP+` option).

52. Violation of MEMORY directive at address $<VALUE>

53. Violation of VARIABLE directive at/near $<VALUE>

ASM11 has produced code and/or data that falls outside any address ranges defined via the `#MEMORY` or `#VARIABLE` directive, respectively. You must either add more `#MEMORY/#VARIABLE` directives to cover the offending range or move your code/data elsewhere (using appropriate segment and/or `ORG` statements).

54. EQU/EXPs require a label, ignoring line

An `EQU` (or `EXP`) by definition is meant to assign a value to a symbol but no symbol name was supplied. Using a repeater value in an `EQU` will also produce this warning for each repetition of the statement except the first one. You should NOT use repeaters with `EQU`.

55. Forward references are always FALSE

Conditional directives other than `#IFDEF` and `#IFNDEF` produce this warning if the symbol(s) referenced have not yet been defined. In this case, the conditional evaluates to false, and if there is an `#ELSE` part, it is taken.

56. String is too long, only first 8 or 16 bits used

8-bit and 16-bit instructions (such as `LDA` and `LDD`) cannot accept a constant string value of more than 8 or 16 bits, accordingly. The longer string encountered is truncated to the first two characters before being used. If an 8-bit operand is expected, you will also get a warning about using a 16-bit value with an 8-bit operand.

57. S19 overlap at address $<VALUE>

The code/data of the shown line overlaps an already occupied memory location at the address shown. The warning appears at code/data that causes the first and consequent overlaps but the problem could be with the original code/data that occupied this address. The assembler has no way of knowing your intentions!

58. RMB overlap at address $<VALUE>

The variable of the shown line overlaps an already defined variable at the address shown. The warning appears at variables that cause the first and consequent overlaps but the problem could be with the original variable that was defined at this address. The assembler has no way of knowing your intentions!

59. Instruction TEST is only valid in SPECIAL TEST MODE

The instruction `TEST` is only defined when running the 68HC11 in special test mode. This warning is generated just in case you meant to say `TST` and typed `TEST` by mistake.

60. Extra operand found ignored

In a `BCLR`/`BSET` you have supplied a branch address. Depending on what you intended to do, either change the instruction to `BRCLR`/`BRSET` or remove the last operand.

61. No ending string delimiter found

The last string quote is missing. ASM11 did its best to produce a value for you but it may not be the one you wanted. For example: «`LDA #'a`» will produce this warning but the value used will be correct, while «`LDA #'a ;comment`» will produce a wrong value (the space after the a because the string `'a '` is a 16-bit value downsize to an 8-bit value, you will get a warning about this also).

62. Operand is larger than 16 bits, using low 16-bits

The operand is greater than 16 bits but the instruction can only accept a 16-bit operand. The lower word was used.

63. Operand is larger than 8 bits, using low 8-bits

The operand is greater than 8 bits but the instruction can only accept an 8-bit operand. The lower byte was used.

64. Possible memory wraparound at address $<VALUE> (<DEC VALUE>)

It seems like you have reached the end of memory ($FFFF) and caused the Program Counter to wrap around to zero. In some situations this may be intentional. Using `RMB 2` (rather than `FDB` or `DW`) in the vector for RESET will also give this warning but it should be ignored. The address shown is the beginning address of the [pseudo-] instruction that caused the wraparound.

65. A JUMP was used when a BRANCH would also work

You could have used a Branch instead of a Jump. This will make your code one byte shorter for each warning. Controlled by the -REL (OPTRELxx) option.

66. Attempting operation with missing first operand

An operation was attempted without an operand before the operator. For example /3 (divide by 3) is missing the dividend.

67. JSR/BSR followed by unlabeled RTS => JMP/BRA

You could safely replace the sequence JSR/RTS or BSR/RTS to a single JMP or BRA, accordingly. The code will remain equivalent but you will gain a byte of memory, two bytes of stack space, and also make it a little faster. It will, however, make your source-code less user-friendly and a bit harder to

follow. It should probably be done only when speed if very important or if you're running out of space and must save every byte you can. WARNING: In certain situations, the code is dependent on the return address pushed on the stack by a JSR or BSR instruction. In those cases, do NOT replace with JMP/BRA because the code will not run correctly. It is assumed you know the code you're working on. Controlled by the -RTS (OPTRTSxx) option.

68. No ORG (RAM:$0000 ROM:$D000 DATA:$B600 VECTORS:$FFD6)

ASM11 started producing code/data without having been told explicitly where to put it. A segment directive may have been used, however, with its default value. NOTE: You will only get this warning once no matter how many segments you are using. This means that you may be required to add ORGs for each segment or else the default values will be used.

69. Phasing on <SYMBOL> (PASS1: $<VALUE>, PASS2: $<VALUE>')

The symbol shown was defined two or more times using different values. The values given may help you determine the type of the problem more quickly, i.e., whether it is a duplicate label with the same purpose or a completely random use of the same symbol name. The assembler will attempt to use the last (most current) value for this symbol.

70. TABSIZE must be a positive integer number, not changed

The `TABSIZE` directive requires a positive integer, and one wasn't supplied. The current tab size was not altered.

71. Unrecognized directive "<DIRECTIVE>" ignored

Something that looks like a directive (i.e., begins with # or $ and appears first in a line after the white-space) was encountered but it wasn't a valid one. Check spelling. If spelling seems correct, you may be assembling someone else's code written for a later version of ASM11 that supports additional directives your version doesn't understand.

72. USER: <USER TEXT>

This is a user generated warning via the #WARNING directive

73. Branching to next instruction is needless

You are using a branch instruction (other than BSR) to send control to the immediately following instruction. This is the default action of the CPU, so this instruction is not required. Controlled by the -REL (OPTRELxx) option.

74. Jumping to next instruction is needless

You are using a Jxx instruction (other than JSR) to send control to the immediately following instruction. This is the default action of the CPU, so this instruction is not required. Controlled by the -REL (OPTRELxx) option.

75. <"Symbol"> symbol size truncated

Automatically generated symbols (e.g., `PROC`-local `@@` and macro-local `$$$` containing symbols) have expanded to a size of more than 19 characters, and this may cause problems with "duplicate symbol" errors. To correct, or avoid this problem, use shorter local symbol names (say, no more than ten characters).

ASM11's Miscellaneous Features

Repeaters

Each opcode or pseudo-opcode may be suffixed by a colon [:] and a positive integer or expression evaluating to a number between 1 and 32767. This is referred to as the <repeater value>. Some examples:

```
        LSRA:4          ;Move high nibble to low

        FCB:256   0     ;Create a table of 256 zeros

        INS:4           ;De-allocate two words from stack
```

Segments

Eight special directives allow you to use segments in your programs. Segments are useful mostly in conjunction with the use of `INCLUDE` files. Since often it is not possible to know the current memory allocation for variables and code when inside a general-purpose `INCLUDE` file, segments help overcome this (and other problems) with ease.

Also, we often want to have our code and data (strings, tables, etc.) grouped in a different way in our source-code than the resulting S19 (object). Segments again give us the ability to have related code, variables, and data together in the source but separated into distinct memory areas in the object file. When using segments, it is common to have a single `ORG` statement for each of the segments, near the beginning of the program. Thereafter, each time we need to «jump» to a different memory segment/area, we use the relevant segment directive.

Although the eight segments are named `#RAM`, `#ROM`, `#EEPROM`, `#XRAM`, `#XROM`, `#DATA`, `#SEGn`, and `#VECTORS` their use is identical (except for the initial default values) and they are interchangeable. Use of segments is optional. If segments aren't used, you are always in the default `#ROM` segment (which explains why code assembles beginning at $D000).

Local Symbols

All symbols that begin with a question mark [?] are considered to be local. Local symbols are local on a per-file basis. Each INCLUDE file (as well as the main file) can have its own locals that will not interfere with similarly named symbols of the remaining participating files. This has two advantages: First, symbols

can be re-used in another INCLUDE file in a completely different way. Second, local symbols are not visible outside the file that contains them. This last benefit makes it possible to write quite complex INCLUDE files while making only the global variables and subroutine entry labels visible to the outside.
Note: You can also have procedure-local symbols. See the `#PROC` *and* `PROC` *directives for details.*

Marking big blocks as comments

#IFDEF without any expression following will always evaluate to False. This can be used to mark out a large portion of defunct code or comments. Simply «wrap» those lines within #IFDEF and #ENDIF directives. This saves you the trouble to individually mark each line as comment, e.g.,

```
#IFDEF

This is a block of comments explaining all the little details of this great assembly language program...
Blah, blah, blah...

#ENDIF
```

The only drawback is that the listing file will not include this section. In some cases this is desirable, in others it isn't.

Creating 'menus' of possible -D option values

ASM11 -Dxxx [[-Dxxx]...] is used to pass up to ten symbols to the program for conditional assembly. Here's a tip for creating 'menus' of possible symbols to use with the -D option, so you don't have to remember them. An example follows:

```
#ifdef ?
     #message ****************************************
     #message * Choice of run-time conditional symbols
     #message ****************************************
     #message * DEBUG: Turns on debugging code
     #message * E2: Target is MC68HC811E2
     #message * E9: Target is MC68HC711E9
     #message ****************************************
     #fatal Run ASM11 -Dx (where x is any of the above)
#endif
```

The command **ASM11 -D? PROGNAME.ASM** will display the above 'menu' of possible -D values and terminate assembly. If you make it a habit of doing this in all your programs, then at any time you're not sure which conditional(s) to use, simply try assembling with the -D? option and you will get help. (A question mark is the smallest possible local symbol you can define. It is a perfect candidate for this job as it is easy to remember because it's like asking for help, and also because it is only visible in the main file. You could, of course, use any other symbol name you like.)

Using -Dx with specific values

You may also assign a specific value to a symbol defined at the command-line. This makes it possible, among other things, to assemble a program at different locations on the fly. For example, the following program:

```
;SAMPLE.ASM
#ifndef ROM ;needed to avoid «Duplicate symbol..» errors
ROM       equ       $F800     ;Default ROM location
#endif
          ORG       ROM
Start     lds       #$FF      ;the program begins here
          ...                 ;rest of program is left to imagination
          bra       *         ;the program ends here
```

will be assembled at $F800 with the command `ASM11 SAMPLE` but you could also assemble with a command similar to this: `ASM11 SAMPLE -DROM:$D000` to move ROM to a different location at assembly time.

As another example, you could declare an array where you define the dimension during assembly. No need to edit the source.

```
;SAMPLE.ASM
#ifndef ARRAYSIZE ;needed to avoid «Duplicate symbol..» errors
ARRAYSIZE equ       10        ;Default size for array
#endif
#if ARRAYSIZE < 2 ;check for minimum size allowed
     #error ARRAYSIZE must be at least 2
#endif
          ORG       RAM
Status    RMB       ARRAYSIZE
Pointer   RMB       ARRAYSIZE*2
          ...

          ORG       ROM
Start     lds       #$FF      ;the program begins here
          ...                 ;rest of program is left to imagination
          bra       *         ;the program ends here
```

Hungarian notation is never used.

`MY_CONSTANTS` are always uppercase and separate words by underscore.

`my_variables` are always lowercase and separate words by underscore.

`_MY_OFFSETS_` are constants surrounded by underscores.

`.my_pointers` are variables beginning with a point (i.e., they point to something)

`MyRoutines` are camelcase names. They may also use underscores mostly to separate module from function name, as in InitializeSCI which could also be written as Initialize_SCI

File local symbols follow the above rules but always start with `?` (question mark).
Example: `?my_variable`

Proc local symbols follow the above rules but always end with `@@`, example: `Loop@@`

Macro local symbols follow the above rules but always end with `$$$`, example `Loop$$$`

Comments always start with a semicolon. Macro comments that do not need expansion always start with a double semicolon.

Labels are defined starting in column 1 and use the default maximum label length of 19 so they remain compatible with P&E map files.

Opcodes, pseudo opcodes (e.g., `ORG`) start in column 21.

Operands start in column 31.

Comments start in column 51 unless a long operand pushes them further to the right.

Procs are separated with an 80-char comment line full of asterisks that always begins with a semicolon.

A non-trivial proc uses a header with Purpose, Inputs, Outputs, and Note(s) that describe the purpose of the proc and the calling interface.

All procs use automatic SP adjustment with the `#spauto` assembler directive.

Procs that refer to caller stack follow `#spauto` with an appropriate offset.
If the proc is near (i.e., ends with an `RTS` instruction) it uses the offset 2.
If the proc is far (i.e., ends with an `RTC` instruction) it uses the offset `:ab` which adjusts automatically based on whether the MMU is used or not.

Each proc begins with the proc name and the assembler directive `proc`.
An `endp` directive is normally not used, unless we need to embed a proc inside another proc for proximity reasons.

Getting rid of the «No ORG...» warning

If you are annoyed by the «No ORG...» warning that shows up whenever the assembler attempts to produce code or data without first having encountered an ORG statement, here's how to turn it off without actually specifying a fixed origin.

Somewhere before any code or data, and regardless of the current segment, use the pseudo-instruction:

```
org  *
```

This is a «No Operation» ORG statement because it will simply use the current location counter for the ORG. It effectively does nothing. It will, however, set the appropriate internal flag that tells the assembler an ORG has been used and, thus, no warning!

Tips on using the MEMORY directive

The MEMORY directive is generally very useful for any program. It could help you save precious debugging time by alerting you whenever you accidentally put code and/or data where there is no real or available memory. The best place to use this directive is the same include file that defines the particulars of a specific MCU or project. And, assuming you always INCLUDE one such file in every program you write, you can forget about it.

Another use for the memory directive is to help you write a program that does not necessarily reside in specific memory locations but, rather, it occupies no more than so many bytes. For example, you're writing a small program that must be no more than 100 bytes long. Here's how to set the MEMORY directive to warn you should you go over this limit:

```
Start     lds       #$FF      ;the program begins here
          ...                 ;rest of program is left to imagination
          bra       *         ;the program ends here

#memory Start Start+100-1     ;allowed range = Start to 100 bytes later
```

If while writing your program you begin getting MEMORY Violation warnings, you'll know you have reached (actually, gone beyond) the allowed limit. You must cut down the size of your code until the warning disappears.

Linux/Win32 version addendum

The DOS/Win and Linux versions are practically identical. This document covers the DOS/Win version.

A few differences with the Linux or Win32 versions are listed below:

The different memory models used with Windows and Linux allow for a far greater number of total symbol definitions. *If while using the DOS version you get "heap memory" issues, try using the Win version, instead.*

Standard error redirection does not work. For Linux all output (not just the errors) is redirected, but this may not be in a very readable format. Please use only the –E option to have ERR files created.

Beginning with v2.00, the Win32 version allows wildcards on the command line for matching multiple assembly language filenames. The Linux version uses standard Linux syntax for multiple filenames. For either version, filenames are not limited to the DOS 8.3 format. The `#INCLUDE` directives within the source code may also specify long filenames. Spaces within long filenames are currently not possible.

For the Win32 or Linux version, you must keep the included asm11.cfg in the same directory as the asm11 binary (for example, ~/bin for Linux or C:\Utils for Win32, etc.). Any time you change options and save them (with the –W option) a new asm11.cfg file will be created in the current directory (or updated if it already exists). If you want to make this new configuration the current directory project's default, leave the .cfg file in that directory, and run asm11 from there.

In case you also want to make this .cfg file the new global default, "mv" (Linux) or "move" (Win32) it to the ~/bin *or other directory* where your asm11 binary is, and anytime a local asm11.cfg isn't found, the global one will be used instead.

Another difference for the Linux version is that filenames are case-sensitive. But, to ease porting from DOS/Win, if a file (e.g., `#INCLUDE`) is not found, it will be searched for again as "all lowercase" and, if not found a second time, it will be searched for once more as "all uppercase." This makes it easier to transfer files from DOS/Win to Linux and not have to rename them, or do so but not have to also change the source code.

These are pretty much the only differences in behavior.

Assembly language source code syntax is identical for all platform versions, except where noted otherwise.

Where to get ASM11

http://www.aspisys.com/asm11.htm

Look for the filename ASM11_84.ZIP where _ maybe replaced with a letter (eg.ASM11D84.ZIP). This is the last fully FREEWARE version. This version will remain FREEWARE and available for public download for good.

Later versions (beginning with 1.85), however, are no longer FREEWARE but will be available for free only for private/non-commercial use.