

ASM8

A two-pass absolute macro cross-assembler for the 68HC08/HCS08/9S08

Quick Reference Guide

ASM8 - Copyright © 2001-2011 by Tony Papadimitriou <tonyp@acm.org>

Latest Update: November 22, 2011 for ASM8 v8.40

Command-Line Syntax and Options

ASM8 [-option [...]] [[@]filespec [...]] [>errfile]

- *option(s)* may appear before, in between or after *filespec(s)*.
- *option(s)* 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. Currently, error redirection is only possible with the DOS version.
- All shaded areas are pertinent to the MMU enabled versions (Win32/Linux only).
- 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 (MMU notwithstanding), 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 (2³²-1). 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 bps rates or 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)

Option	Default	Description
-C[±]	-C-	Label case sensitivity: + = case sensitive (See also #CASEON/#CASEOFF)
-Dlabel [:expr]		Use up to ten times to define symbols for use with conditional assembly (IFDEF and IFNDEF directives). 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 be used as an #INCLUDE file for other programs. This allows for automatic creation of include files with global exported symbols.
-FD [±]	-FD-	<p>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.</p> <p>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.</p> <p><i>WARNING: Do NOT include this option in batch or makefiles that compile your programs automatically, or you risk producing consistently mis-dated firmware. It should only be used for manual verification purposes.</i></p> <p>It's not by accident this option cannot be saved with the -w switch.</p>
-HCS [±]	-HCS-	When on, the assembler understands the extended HCS08 instruction set. The cycle counts in the listing also reflect the HCS08 core. To check the current status of this switch look at the help screen's second line from top. The software will say it's either a MC68HC08 or a MC68HCS08 assembler based on this setting. See also the directives #HCSON , #HCSOFF , #IFHCS , and #IFNHCS
-Ix		Define default INCLUDE directory root. Relative path files not found relative to the main file, will be tried next relative to this directory. This switch does not affect absolute path file definitions. Affects both the INCLUDE and the IF(N) EXISTS directives.
-J [±]	-J+	<u>Effective only while the MMU is disabled:</u> When on, CALL/RTC instructions are treated as if they were JSR/RTS instructions, respectively. When off, CALL/RTC instructions produce errors. <i>Makes it possible to write common library functions using CALL/RTC instead of JSR/RTS and have them used in all MCUs, regardless if they have an MMU or not.</i> See also the directives #JUMP , and #CALL
-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.
-LS [\pm]	-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 compatible format (possibly not useful for HC08), or N for NoICE SYM format.
-M [\pm]	-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.
-MMU [\pm]	-MMU-	Enable the MMU features (e.g., CALL/RTC instructions, 24-bit addresses and expressions). See also the directives #MMU, #NOMMU, #IFMMU, and #IFNOMMU
-MTx	-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 [\pm]	-O+	Enables these three warnings: 'S19 overlap', 'Violation of MEMORY directive', and 'Violation of VARIABLE directive'.
-P [\pm]	-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 do not affect this.
-Q [\pm]	-Q-	Specifies quiet run (no output) when redirecting to an error file (DOS only). Useful for IDEs that call ASM8 and don't want to have their display messed up. Beginning with v1.29, 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 <i>n</i> 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 while maximum is 250.
-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 [\pm]	-REL+	Allows generation of «BRA/BSR instead of JMP/JSR» optimization warnings when enabled. (See also OPTRELON/OPTRELOFF)
-RTS [\pm]	-RTS-	Allows generation of «JSR followed by RTS» subroutine call optimization warnings when enabled. (See also OPTRTSON/OPTRTSOFF)
-S [\pm]	-S+	Generate *.S19 object file (one for each file assembled).
-S2 [\pm]	-S2-	Force the generation of S2 records (24-bit addresses) even for 16-bit addresses. Although 24-bit addresses are enabled, no MMU features are enabled. Useful mostly for forcing 16-bit addresses to appear as 24-bit (with leading byte as \$00) so that S19 loaders can use that as the PPAGE value. See also #S1 and #S2
-S9 [\pm]	-S9+	This option can be used to turn off generation of the final S9 (or S8) 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.

		<p>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. <i>This option cannot be saved with the <code>-w</code> switch.</i></p> <p><i>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 <code>-s9</code> option in effect.</i></p>
-SH [\pm]	-SH-	Include dummy «S0» record (header) in object file (only if -s+).
-SP [\pm]	-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 [\pm]	-T-	Makes all errors look like Borland errors (useful to fool certain IDEs).
-T_n	-T8	Specifies tab field width to use in *.LST files. Tab characters embedded in the source file are converted to spaces in the listing file such that columns are aligned to 1 + every <i>n</i> th character.
-U_x		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. <i>Not available in the DOS version.</i>
-X [\pm]	-X+	Allow recognition of extra, non-68HC08-standard mnemonics and simulated index modes in source files. (See also EXTRAON/EXTRAOFF)
-Z [\pm]	-Z-	Convert the paged addresses (<code>PAGE:ADDR16</code>) to linear (extended) address in the produced S19 file(s). In effect, all addresses within the ranges <code>\$xx8000-\$xxBFFF</code> are converted to their linear format. The code or listing is not affected at all. This is useful for S19 loaders that expect addresses in linear format, instead of paged format. <u>Warning:</u> The ambiguous case of <code>\$8000-\$BFFF</code> is treated as <code>PAGE0</code> , which is converted to linear addresses: <code>\$000000-\$003FFF</code> . If you want to place something at <code>PAGE2</code> , position it (<code>ORG</code>) at <code>\$028000</code> , which will convert to linear address <code>\$008000</code> .
-WRN [\pm]	-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 <code>-REL</code> and <code>-RTS</code>).
-W	(none)	Write options specified on command line to the ASM8 executable (DOS), or ASM8.CFG file (Win/Linux). The user-specified options become the default values used by ASM8 in subsequent invocations. <i>Filespec(s)</i> on the command line are ignored. Assembly of source files does not take place if this option is specified.

Source File Pseudo-Opcodes (Pseudo-Instructions)

All shaded areas are pertinent to the MMU enabled versions (Win32/Linux only).

Pseudo-Op	Description
<i>[label]</i> ALIGN <i>expr</i>	<p>Case 1. If no label is present, it aligns the current location counter to be a multiple of the given expression.</p> <p>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.</p> <p><u>COMPATIBILITY ISSUE WITH VERSIONS PRIOR TO 8.30:</u></p> <p>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.</p>
DB <i>string expr[,...]</i>	Define Byte(s). <i>expr</i> 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 <i>expr</i> encountered (using the LSB of the result) or one byte for each character in <i>strings</i> .
DS <i>blocksize</i>	Define Storage. The assembler's location counter is incremented by <i>blocksize</i> . Forward references not allowed. No code is generated.
DW <i>expr[,...]</i>	Define Word(s). <i>expr</i> may be a constant numeric, a label or an expression. <i>expr</i> 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 [<i>expr</i>]	<p>Provided for compatibility. The END directive cannot be used to terminate assembly; ASM8 always processes the source file to the end of file. If <i>expr</i> is specified, the word result of the final END directive is encoded in the S9 record of the object file.</p> <p>If the <i>expr</i> specified is 24-bit (bits 23-16, collectively, are non-zero), which is possible only when the MMU option is enabled, the 24-bit result is encoded in the S8 record of the</p>

	object file (no S9 record is produced in that case).
ENDM	Ends definition of a macro.
<i>label</i> DEF <i>expr</i>	<p>Assigns a DEFault value to a label. <i>In other words, this is a conditional EQU. It only assigns the label if the label is currently undefined.</i></p> <p>LABEL DEF EXPR</p> <p>is equivalent to:</p> <pre>#IFNDEF LABEL LABEL EQU EXPR #ENDIF</pre> <p>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.</p>
<i>label</i> EQU <i>expr</i>	Assigns the value of <i>expr</i> to <i>label</i> . See also EXP and SET
<i>label</i> EXP <i>expr</i>	Assigns the value of <i>expr</i> to <i>label</i> . 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.
FAR <i>expr</i> [, ...]	<p>Define 24-bit word(s) when the MMU is enabled. <i>expr</i> may be a constant numeric, a label or an expression. <i>expr</i> is always interpreted as a 24-bit quantity, and is stored in the object file at the current location counter in big-endian order.</p> <p><i>If, however, the MMU option is disabled, FAR is treated as DW.</i></p>
FCB <i>string</i> <i>expr</i> [, ...]	Form Constant Byte(s). Same as DB .
FCC <i>string</i> <i>expr</i> [, ...]	Form Constant Character(s). Same as DB .
FCS <i>string</i> <i>expr</i> [, ...]	Form Constant String. Similar to FCC , but automatically appends a terminating null (0) byte to the end of the string defined (for ASCIZ strings).
FDB <i>expr</i> [, ...]	Form Double Byte(s). Same as DW .
LONG <i>expr</i> [, ...]	<p>Form 32-bit long word(s). <i>expr</i> may be a constant numeric, a label or an expression. <i>expr</i> is always interpreted as a 32-bit quantity, and is stored in the object file at the current location counter in big-endian order.</p> <p><i>Not available in the DOS version.</i></p>
<i>MacroName</i> MACRO <i>comments</i> ... 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. REMACRO is same as MACRO when used with special ? macro.*

- 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.
- 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.
- 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.
- 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.
- 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.
- 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.

- 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.
- Local macro names start with the `?` symbol (like it is done with normal local labels).
- 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.)
- 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.)
- 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 except 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 non-trailing 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).

- 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
- 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.)
- 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).
- 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).
- 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

	<p>use). With conditional assembly (e.g., #IFPARM ~#~) one can treat the ~1~ parameter differently, assuming immediate mode.</p> <ul style="list-style-type: none"> ▪ 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). <i>Note: If no # is inside the parameter, ~#n~ is the same as ~n~ alone.</i> ▪ 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). ▪ 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 <u>chained</u> (or <u>nested</u>) 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`.*

- 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.
- 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).
- 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`).
- 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`
- Similarly, the placeholder `~#text~` (case-insensitive) returns the part after a possible `#` symbol (if one is present).
- 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

	<p>the macro definition is located (<code>~mfilename~</code> etc.), which is not necessarily in the current file.</p> <ul style="list-style-type: none"> ▪ The placeholder <code>~@~</code> is an alias for the full list of placeholders separated by <code>•</code> (starting from <code>~1~</code>). Useful if you want to pass all parameters to another macro. The sequence produced by <code>~@~</code> is: <code>~1~•~2~•~3~•~4~•~5~•~6~•~7~•~8~•~9~</code> ▪ The placeholder <code>~@@~</code> is an alias for the full list of placeholders separated by <code>•</code> but starting from <code>~2~</code>. 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 <code>~@@~</code> is: <code>~2~•~3~•~4~•~5~•~6~•~7~•~8~•~9~</code> ▪ Trailing parameter separators (commas by default) <u>and</u> trailing commas due to macro expansion of null parameters are automatically removed. This is particularly useful when writing macros, which replace or enhance <u>single-operand</u> 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: <code>LDA ~1~, ~2~</code> will work even if <code>~2~</code> is null, because the now 'dangling' comma after <code>~1~</code> will be automatically removed, preventing an otherwise expected syntax error. Similarly, <code>LDA ~1, ~+1~, 1~, ~2~</code> 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. ▪ You can use the <code>~n.s.l~</code> format of the parameter placeholder (where n is the parameter number, or the case-insensitive keyword <code>label</code>, 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 <code>s</code> position) to extract only a portion of the text of the corresponding parameter or constant. The first dot is required (to disambiguate from <code>~n~</code> type parms) even if nothing follows. The <code>s</code> and <code>l</code> are optional. If <code>s</code> is not entered its value is assumed to be one, so that <code>~1.~</code> is the same as <code>~1~</code> alone. If <code>l</code> is not entered, its value is the length from <code>s</code> to the end of the parameter (i.e., the remaining string). Note: The assembler forces <code>s</code> and <code>l</code> 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.

- Order of placeholder expansion is: ~@~, ~@@~, ~label~, ~macro~, ~00~, ~self~, ~text~, ~#~, ~#n~, ~n~ (where n = 0..9, in that order), \, , and •, {expression}, ~n.s.l~, and \@string\@.
- 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 interpreted 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.

- Macros cannot `#INCLUDE` files, but can 'chain' to one.
- Macros cannot define other macros.
- 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'.*)
- Macros can 'chain' to self or other macros (with no automatic return). This allows, among other things, for creating loops, making macros very powerful.
- 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`***

	<p><i>@@ mode (see the relevant section for details) . Macro-chaining will be altogether disabled, however.</i></p> <ul style="list-style-type: none">▪ <i>To break out of an accidental endless macro loop, press [ESC] on the command-line.</i>▪ <i>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.</i>▪ <i>Virtually unlimited number of macro definitions (memory permitting.)</i>▪ <i>Virtually unlimited size of each macro (memory permitting.)</i>▪ <i>Unlimited number of macro invocations (all internal macro counters are 32-bit).</i>
--	--

MERROR [text]	Combines an #ERROR directive followed immediately by an MEXIT, which is commonly found in macros. <i>This can only be used inside macros.</i>
MEXIT [expr]	<p>Causes an unconditional early exit from a macro expansion. (Normally, used inside a conditional block.)</p> <p>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.</p>
MSUSPEND MRESUME	<p>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.</p> <p>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.)</p> <p>This makes it much easier to create <i>nestable</i> 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.</p> <p><i>Note: When a nested macro is suspended, <u>all</u> macros leading to the currently executing macro are indirectly suspended as a side effect.</i></p> <p>MRESUME can be used only from outside any macros to resume execution of the most recently suspended macro.</p> <p>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.</p> <p>The recursion limit (see #MLimit) counts suspended macros also, because these are stacked just like when doing normal nested macro calls.</p>

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#]	<p>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.)</p> <p>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).</p>
MSTR index[,index]*	<p>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.</p> <p>It is equivalent to the following sequence (but repeated for each specified index n):</p> <pre>#IFPARM ~n.~ #IFNOSTR ~n.~ MSET n, \@~n.~\@ #ENDIF #ENDIF</pre>
MSET index[,text] MDEF index[,text] MSWAP index,index	<p>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.</p> <p>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).</p> <p>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.</p> <p><i>Note: index is any expression that doesn't contain forward references.</i></p>
MREQ ind[,ind]*[:errmsg]	<p>Checks each of the specified macro parameters (separated with commas) for null value (empty). If the parameter is null,</p>

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`.

<p>MTOP [limit expr]</p>	<p>Causes an immediate <i>unconditional</i> jump to the top line of the current macro, while incrementing the <code>:LOOP</code> counter. It can be used either alone or within conditionals. The advantage to using <code>MTOP</code> over <code>@~0~</code> (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 <code>:LOOP</code>. This means, however, that <code>\$\$\$</code> based labels (<i>which are unique to a macro invocation</i>) are still in the same scope as before the <code>MTOP</code> command since no new macro has been invoked.</p> <p>If the optional limiting expression (containing only non-forward references) is present, its value will be compared to the <code>:LOOP</code> counter and <code>MTOP</code> will execute only if the current value of <code>:LOOP</code> is less than the value of the expression.</p> <p>Example (shift word right one or more times):</p> <pre> lsr.w macro Address[,Count] mdef 2,1 ;default Count=1 lsr ~1~ ror ~1,~+1~,1~ mtop ~2~ endm </pre> <p>As another example, an expression like the one that follows can be used to loop while the next parameter is not null:</p> <pre> mtop :loop+:{:loop+1} </pre>
<p>MDO [start expr] MLOOP [limit expr]</p>	<p><code>MDO</code> and <code>MLOOP</code> work together to form a local <code>DO ... LOOP</code> inside a macro. <i>Note: <code>MDO</code> and <code>MLOOP</code> cannot be nested because <code>MLOOP</code> always matches the most recent <code>MDO</code> of the current macro.</i></p> <p><code>MDO</code> simply marks the current line (i.e., the line containing the <code>MDO</code> keyword) as the beginning of a local loop, and (re)initializes the <code>:MLOOP</code> counter to one (1), or to the value of the non-forward expression, if one is present.</p> <p><code>MLOOP</code> causes an immediate <i>unconditional</i> jump to the line following the most recent <code>MDO</code> keyword, while incrementing the <code>:MLOOP</code> counter (not to be confused with the <code>:MACROLOOP</code> or <code>:LOOP</code> counter). If no <code>MDO</code> was used up to this point in the macro, <code>MLOOP</code> jumps to the top of the current macro (just like <code>MTOP</code> would), but it only affects the <code>:MLOOP</code> counter (whereas <code>MTOP</code> only affects the <code>:LOOP</code> counter).</p>

	<p>If the optional limiting expression (containing only non-forward references) is present, its value will be compared to the <code>:MLOOP</code> counter and <code>MLOOP</code> will execute only if the current value of <code>:MLOOP</code> is less than the value of the expression. Example (multi-byte addition):</p> <pre> add.m macro Op1,Op2,Ans[,Size] mdef 4,1 ;default size = 1 #push #spauto :sp psha mdo lda ~1,~+{~4~-:mloop}~,1~ #if :mloop = 1 add ~2,~+{~4~-:mloop}~,2~ #else adc ~2,~+{~4~-:mloop}~,2~ #endif sta ~3,~+{~4~-:mloop}~,3~ mloop ~4~ pula #pull endm </pre> <p>As another example, an expression like the one that follows can be used to loop while the next parameter is not null: <code>mloop :mloop+:{:mloop+1}</code></p>
<code>label NEXP symbol[,expr]</code>	<p>Assigns the <u>current</u> value of <i>symbol</i> to <i>label</i> as if with <code>EXP</code>. Then, it increments the value of <i>symbol</i> by one (as if with <code>SET</code>) 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: <i>symbol</i> is a single label and not an expression. See also NEXT, SETN</p>
<code>label NEXT symbol[,expr]</code>	<p>Assigns the <u>current</u> value of <i>symbol</i> to <i>label</i> as if with <code>EQU</code>. Then, it increments the value of <i>symbol</i> by one (as if with <code>SET</code>) 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: <i>symbol</i> is a single label and not an expression. See also NEXP, SETN</p>
<code>ORG expr</code>	<p>Sets the assembler's location counter for the active segment. Code generated after this directive will be assembled starting at the location specified by <i>expr</i>.</p>
<code>label PROC</code>	<p>First, it advances the @@ local label counter, and then it assigns the value of the program counter (*) to <i>label</i>. This allows using symbols locally for a specific section of code</p>

	<p>(e.g., a subroutine). The symbol to the left of <code>PROC</code> is always in the new scope. Each time <code>PROC</code> (or <code>#PROC</code>) is encountered, the assembler increments an internal 32-bit local symbol counter. Symbols containing <code>@@</code> anywhere inside their name (except at the very beginning) <i>at least once</i> (for example, <code>Loop@@</code>) will have the <code>@@</code> part replaced with a special control character (different from what is used with macro local <code>\$\$\$</code>) and the current value of the internal local symbol counter (similar to <code>\$\$\$</code> with macro local labels).</p> <p>Up until a <code>PROC</code> or <code>#PROC</code> is encountered in the program, the <code>@@</code> is not treated specially (i.e., the <code>@@</code> 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 <code>:PROC</code></p> <p>See also #PROC</p>
RMB <i>blocksize</i>	Reserve Memory Byte(s). Same as DS .
<i>label</i> SET <i>expr</i>	<p>Assigns the value of <i>expr</i> to <i>label</i> even if <i>label</i> is already defined with a different value.</p> <p>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.</p> <p><i>Warning:</i> 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.</p> <p>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.</p> <p><i>Example of wrong use:</i></p> <pre> 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 </pre> <p>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).</p> <p><i>Example of correct use:</i></p> <pre> 1. Value equ 123 2. lda #Value ;we expect 123, actual is 123 ... 3. Value set 234 </pre>

	<p>4. <code>lda #Value ;we expect 234, actual is 234</code> See also EXP and EQU</p>
<p><i>label</i> SETN <i>symbol</i> [<i>,expr</i>]</p>	<p>Assigns the <u>current</u> value of <i>symbol</i> to <i>label</i> as if with SET. Then, it increments the value of <i>symbol</i> 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: <i>symbol</i> is a single label and not an expression. See also NEXP, NEXT</p>

Source File Processing Directives

- All processing directives must be prefixed with a \$ or # character. ASM8 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 spaces display, use the Alt-255 character, as many times as needed.
- All shaded areas are pertinent to the MMU enabled versions (Win32/Linux only).

Directive	Description
#AIS <i>symbol</i>	<p>#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 (<i>note: a plain symbol, not an expression</i>), indicating a possible stack frame definition error (assuming correct placement of the relevant directives).</p> <p>The warning also shows the correct AIS instruction that is required to correct the problem.</p> <p>This directive makes it very easy to correct the numeric value in AIS instructions to match the <u>following</u> 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 AIS instruction using a one-based starting offset just so you can use a label with AIS and then having to re-define it for dynamic assignment of offsets based on the current :SP.</p> <p><i>The associated :AIS symbol returns the difference between the current :SP and the value saved during the most recent AIS instruction. 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 (inclusive of the previous AIS instruction). This is only meant for use in #SPAUTO modes, which automatically adjusts the current value of the :SP internal symbol.</i></p> <p>#PUSH and #PULL will save/restore the value of this setting.</p>

	<p>Example use:</p> <pre> #spauto Subroutine ais #-4 ;local data ? ?Parm1 setn ? ,2 ?Parm2 setn ? ,2 #ais ? ;check frame definition </pre>
<p>#CALL</p>	<p>Effective only while the MMU is disabled: When active, CALL/RTC instructions are NOT treated as if they were JSR/RTS instructions, but they issue errors instead. See also the directives #JUMP, #MMU, #NOMMU Equivalent to the -J- command line option.</p>
<p>#CASEOFF</p>	<p>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. Equivalent to the -C- command line option.</p>
<p>#CASEON</p>	<p>When #CASEON is in effect, symbol references are NOT internally converted to uppercase before they are searched for or placed in the symbol table. Equivalent to the -C+ command line option.</p>
<p>#CRC <i>expr</i></p>	<p>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.</p> <p>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).</p> <p>The computed CRCs are available by accessing the internal symbols :CRC and :S19CRC</p> <p>The formula used for the 16-bit CRC calculation is very simple to be easily implemented even in tiny bootloaders:</p> $\mathbf{16BitCRC := 16BitCRC + 16BitAddress * 8BitData}$ <p>: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.</p>

	<p><code>:CRC</code>, on the other hand, is mostly useful for checking code after it has been loaded into the MCU, at each reset, for example.</p> <p>Please note that for both CRCs all <code>\$00</code> bytes do not affect the calculation while, for the user CRC only (<code>:CRC</code>), all <code>\$FF</code> 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. <i>As a side effect, however, any <code>\$00->\$FF</code> or <code>\$FF->\$00</code> alterations in the file cannot be detected with the user CRC.</i></p>
#CYCLES [<i>expr</i>]	<p>First, the optional expression is calculated using the current values of any internal symbols.</p> <p>Then, the current value of <code>:CYCLES</code> is copied to <code>:OCYCLES</code>.</p> <p>Finally, the internal <code>:CYCLES</code> counter is set to zero (if the optional expression is missing), or to any arbitrary value (the result of the expression).</p> <p><i>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.</i></p>
#DATA	<p>Activation of the DATA segment. Default starting value is <code>\$FE00</code>.</p>
#DROP macro[,macro]*	<p>Undefines one or more macros. If a macro is not currently defined, a warning will be issued (to protect from possible typing errors).</p> <p>To drop all macros (global and local) with a single command, use <code>*</code> (asterisk) in place of the macro name. <i>There is no warning if no macros found.</i></p> <p>To drop all local macros (for the current file only) with a single command, use <code>?*</code> (question mark followed by asterisk) in place of the macro name. <i>There is no warning if no local macros found.</i></p> <p>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.</p> <p>The special macro named <code>?</code> (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</p>

	<p>#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.</p> <p>You cannot drop macros that are currently active above the current macro level (e.g. nested macros leading to current one.)</p>
#EEPROM	Activation of the EEPROM segment. Default starting value is \$0000.
#EJECT	See #PAGE
#ELSE	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.
#ENDIF	Marks the end of a conditional-assembly block. Conditional assembly statements may be nested if they are properly blocked with #ENDIF directives.
#ERROR [<i>text</i>]	When encountered in the source, the assembler issues an error message in the same form as internally-generated errors, using the <i>text</i> specified, prefixed with «USER: »
#EXIT [<i>expr</i>]	<p>If no expression is present, it immediately exits the current #INCLUDE file. (<i>Does nothing if used inside a main file.</i>)</p> <p>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).</p> <p>This can be used in the top of #INCLUDE files, like so:</p> <pre> #EXIT _COMMON_ __COMMON__ </pre> <p>In this example, the first time this file is included, the symbol <code>_COMMON_</code> is undefined, so the #EXIT is ignored. Consequent times this file is included, it exits upon hitting the #EXIT directive.</p> <p><i>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)</i></p>
#EXTRAOFF	Disables recognition of ASM8's extended instruction set for source lines that follow this directive.

	Equivalent to the -X- command line option.
#EXTRAON	Enables recognition of ASM8's extended instruction set for source lines that follow this directive. Equivalent to the -X+ command line option.
#EXPORT <i>symbol</i> [, <i>symbol</i>] *	Export one or more symbols (as if with <code>EXP</code>). File-local symbols cannot be exported. If a symbol is not currently defined, a warning will be issued.
#FATAL [<i>text</i>]	Similar to the #ERROR directive, but generates an assembler fatal error message and terminates the assembler (possible further files in the list will not be processed).
#HCSOFF	Disables the HCS08 instruction set mode. See also #IFHCS #IFNHCS #HCSON Equivalent to the -HCS- command line option.
#HCSON	Enables the HCS08 instruction set mode. See also #IFHCS #IFNHCS #HCSOFF Equivalent to the -HCS+ command line option.
#HOMEDIR [<i>path</i>]	Makes the specified <i>path</i> the current home directory. Although this cannot affect where any output files will go, it does make a difference on where any following <i>relative</i> #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 [<i>path</i>] is missing, the original main file path is restored.
#IF <i>expr1 cond expr2</i>	Evaluates <i>expr1</i> and <i>expr2</i> (which may be any valid ASM8 expression) and compares them using the specified <i>cond</i> conditional operator. If the condition is true, the code following the #IF operator is assembled, up to its matching #ELSE or #ENDIF directive. <i>Cond</i> may be any one of: < <= = >= > <> The condition is always evaluated using unsigned arithmetic. If a symbol referenced in <i>expr1</i> or <i>expr2</i> is not defined, the statement will always evaluate as false.
#IFDEF <i>expr</i>	Attempts to evaluate <i>expr</i> , 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 <i>expr</i> must be defined before the directive for the result to evaluate true (e.g., forward references will evaluate as false). #IFDEF without an <i>expr</i> following will always evaluate to False.
#IFEXISTS <i>fpath</i>	Checks for the existence of the file specified by <i>fpath</i> (using the same rules as those used for #INCLUDE directives) and assembles the code that follows if the specified <i>fpath</i> exists.

#IFHCS	Assembles the following code if the assembler is in the extended HCS08 instruction set mode. See also #IFNHCS #HCSON #HCSOFF
#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 <i>macro</i> #IFNOMDEF <i>macro</i>	#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.
#IFMMU	Assembles the code that follows if the MMU option is enabled. See also the directives #MMU , #NOMMU , and #IFNOMMU
#IFNOMMU	Assembles the code that follows if the MMU option is disabled. See also the directives #MMU , #NOMMU , and #IFMMU
#IFPARM <i>text</i> [= <i>text</i>] #IFPARM <i>text</i> == <i>text</i> #IFNOPARM <i>text</i> [= <i>text</i>] #IFNOPARM <i>text</i> == <i>text</i> <i>Aliases:</i> #IFB same as #IFNOPARM #IFNB same as #IFPARM	Normally used inside macros. If <i>text</i> 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 <i>text</i> following (after macro expansion) will always evaluate to False. <i>text</i> 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 <i>text</i> 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 (<i>automatic SP adjustment</i>) mode. See also #SPAUTO #SP

<pre>#IFSTR text #IFNOSTR text</pre>	<p>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. <i>text</i> is usually a parameter placeholder (e.g., ~1~).</p> <p>#IFNOSTR performs the opposite test.</p>
<pre>#IFNUM text #IFNONUM text</pre>	<p>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. <i>text</i> is usually a parameter placeholder (e.g., ~1~).</p> <p>#IFNONUM performs the opposite test.</p>
<pre>#INCLUDE fpath #USES fpath</pre>	<p>Includes the specified <i>fpath</i> file in the assembly stream, as if the 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 <i>fpath</i> specifications are always referenced to the directory in which the main source file resides, including any relative #INCLUDE <i>fpath</i> references in nested include files.</p> <p>#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., <i>an analog temperature sensor driver module #USES the A/D driver module, but not the other way around</i>). 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</p> <pre>#IFNDEF _MODULE_ _MODULE_ ...your module code goes here... #endif</pre> <p>technique used to prevent multiple inclusions of the same</p>

	<p>file, but only have it included the first time it is referenced. Normally, the <code>#IFDEF ... #ENDIF</code> block is found inside the file, meaning the assembler must enter the file before it 'knows' it doesn't need it. The advantage with <code>#USES</code>, 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 <code>#INCLUDE</code> files.)</p> <p>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.</p> <p>Also, you cannot use <code>#USES</code> in place of <code>#INCLUDE</code> for modules that must be included multiple times (e.g., including the same SCI driver module, once for each hardware SCI available), <i>although you could use <code>#USES</code> to include a file that itself does <code>#INCLUDE</code> the same file multiple times.</i></p> <p>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.</p>
#IFDEF <i>expr</i>	Evaluates <i>expr</i> 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 <i>fpath</i>	The opposite of #IFEXISTS ; code following this directive is assembled if the specified <i>fpath</i> does NOT exist. The <code>-lx</code> directory will be used also to determine if a file exists or not.
#IFNHCS	Assembles the following code if the assembler is in the regular HC08 instruction set mode. See also #IFHCS #HCSON #HCSOFF
#IFNZ <i>expr</i>	Evaluates <i>expr</i> and assembles the code that follows if the expression evaluates to a non-zero value. #IFNZ always evaluates to false if <i>expr</i> references undefined or forward-defined symbols.
#IFTOS <i>expr</i>	If top-of-stack evaluates <i>expr</i> +:SP (+:SP is implied) and

	<p>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 <i>expr</i> references undefined or forward-defined symbols. Useful mostly in #SP[AUTO] modes.</p>
#IFZ <i>expr</i>	<p>Evaluates <i>expr</i> and assembles the code that follows if the expression is equal to zero. #IFZ always evaluates to false if <i>expr</i> references undefined or forward-defined symbols.</p>
#JUMP	<p><u>Effective only while the MMU is disabled:</u> When active, CALL/RTC instructions are treated as if they were JSR/RTS instructions, respectively. <i>Makes it possible to write common library functions using CALL/RTC instead of JSR/RTS to be used in any MCU, regardless if an MMU is available/used or not.</i> See also the directives #CALL, #MMU, #NOMMU Equivalent to the -J+ command line option.</p>
#LISTOFF #NOLIST	<p>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.</p>
#LISTON #LIST	<p>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).</p>

#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 ("Macros Come First") 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

	<p>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 <code>!instruction</code> format.</p> <p>#@MACRO turns off this option. This is the <u>default</u> setting when a new assembly begins. In this mode, you can only invoke macros with the <code>@</code> prefix. This is the recommended mode for most normal applications.</p> <p><i>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 <code>@macro</code> (or <code>%macro</code>) 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 [<code>@</code> and <code>%</code>].</i></p> <p>Note: If the optional <code>@@</code> 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 <code>@@macro</code> 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-<code>@@</code> macro calls include an implied following <code>mexit</code>). To simulate the same behavior, when the <code>@@</code> option is active, make sure you add an <code>MEXIT</code> command after each otherwise 'chained' macro call. By the way, this will make the macro work the same way regardless of the <code>@@</code> sub-mode being in effect or not. When the <code>@@</code> 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 <code>@@</code> sub-mode, simply give any of these directives without it.</p>
<p>#MEXPORT <code>macro[,macro]*</code></p>	<p>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.</p>
<p>#MLIMIT <code>[expr]</code></p>	<p>Sets the maximum macro nesting limit to the value of the optional expression.</p> <p>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).</p> <p><i>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 (<code>MTOP</code> instruction).</i></p>

#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	<p><i>Note: These two directives work only when the -LC- (List Conditionals = OFF) command-line option is in effect.</i></p> <p>#HideMacros treats all macro-specific keywords (the <code>@macro</code> call, <code>mexit</code>, <code>mtop</code>, <code>endm</code>) 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.</p> <p><i>Note: The corresponding macro definitions will not display at all, regardless of the -LC mode.</i></p> <p>#ShowMacros (re-)enables normal display. The default setting when a new assembly begins is #SHOWMACROS.</p> <p>#PUSH and #PULL will save/restore the value of this setting.</p>
#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 <code>addr1</code> [<code>addr2</code>] #MEMORY <code>#OFF#</code>	Maps a memory location (or range, if <code>addr2</code> 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. <code>Addr1</code> and <code>addr2</code> may be specified in any order. The range defined will always be between the smaller and the higher values. The special keyword <code>#OFF#</code> removes all current definitions. See also #VARIABLE
#MESSAGE [<code>text</code>]	Displays <code>text</code> on screen during the first pass of assembly

	when this directive is encountered in the source. Messages are not written to the error file. They are meant to inform the user of the options used or conditional path taken.
#MMU	Enable the MMU features (e.g., <code>CALL/RTC</code> instructions, 24-bit addresses and expressions). See also the directives #NOMMU , #IFMMU , and #IFNOMMU Equivalent to the <code>-MMU+</code> command line option.
#NOMMU	Disable the MMU features (e.g., <code>CALL/RTC</code> instructions, 24-bit addresses and expressions). See also the directives #MMU , #IFMMU , and #IFNOMMU Equivalent to the <code>-MMU-</code> command line option.
#NOWARN	Turns warnings off. Equivalent to the <code>-WRN-</code> command line option. See also #WARN
#OPTRELOFF	Disable «BRA/BSR instead of JMP/JSR» optimization warnings. Equivalent to the <code>-REL-</code> 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 <code>-REL+</code> command line option.
#OPTRTSOFF	Disable RTS-after-JSR/BSR optimization warning (default). Equivalent to the <code>-RTS-</code> 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 <code>-RTS+</code> does the same thing.
#PARMS [<i>char</i> <i>SPACE</i>]	Allows changing the delimiter used to separate macro parameters when invoking the macro. If <i>char</i> is defined the new delimiter will be the same as <i>char</i> . If there is no character following the directive, the default parameter delimiter (<i>a comma</i>) will be used. To use a regular space as a parameter separator, the [<i>char</i>] part of the command should be the special keyword <code>SPACE</code> (case-insensitive). #PUSH and #PULL will save/restore the value of this setting.
#PPC	#PPC (stands for P reserve P C) simply keeps a copy of the current <code>:PC</code> value to be used later by the <code>:PPC</code> internal symbol. #PUSH and #PULL will save/restore the value of this setting.
#PROC	Advances the @@ local label counter. See also PROC
#PSP	#PSP (stands for P reserve S P) simply keeps a copy of the current <code>:SP</code> value to be used later by the <code>:PSP</code> internal symbol.

	<p>The <code>:PSP</code> symbol returns the difference between the then current <code>:SP</code> 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 <code>#SPAUTO</code> mode, which automatically adjusts the current value of the <code>:SP</code> internal symbol.</p> <p><code>#PUSH</code> and <code>#PULL</code> will save/restore the value of this setting.</p>
<p>#RENAME oldname, newname #REMACRO oldname, newname</p>	<p>Renames a macro from its current (old) name to a new name.</p> <p>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.</p> <p><code>#REMACRO</code> is the same as <code>#RENAME</code> except that it does NOT check if the new name exists. If it exists, there will now be one extra instance of that macro. <i>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.</i></p> <p>Tip: An example of where <code>#RENAME</code> 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:</p> <p>[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.</p> <p>[2] Alter the library macro to include the new behavior. Problem: Other applications may not like the new behavior.</p> <p>[3] Use <code>#RENAME</code> 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.</p>

	<p>Example for #REMACRO that allows front-ending a previous macro to add code before and after the macro call.</p> <pre> 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 </pre>
#S19FLUSH	<p>Forces the immediate termination of an S-record line when encountered, rather than waiting for the record to reach the size specified by the -Rn command line directive. This directive may be used to make identification of the end of code blocks easier when viewing the *.S19 file.</p>
#S1 #S2	<p>#S2 forces the generation of S2 records (24-bit addresses) even for 16-bit addresses. (#S1 reverts to normal, which produces S1 or S2 records based on address.) Although 24-bit addresses are enabled, no MMU features are enabled. Useful mostly for forcing 16-bit addresses to appear as 24-bit (with leading byte as \$00) so that S19 loaders can use that as the PPAGE value.</p> <p>Equivalent to the -s2- and -s2+ command line options.</p>
#SP [expr] #SP1 [expr] #SPAUTO [expr] #SPADD [expr]	<p>#SP1 automatically adds one to all SP indexed offsets. It does this without affecting the current value of the :SP internal symbol.</p> <p>#SP without any expression cancels #SP1 and #SPAUTO modes (reverts to default/normal operation).</p> <p>#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.</p>

	<p>#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.</p> <p>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)!</p> <p>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.</p> <p>#SPAUTO 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.)</p> <p>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:</p> <pre>#SPADD LOOPCOUNT-1</pre> <p>#PUSH and #PULL will save/restore the current setting of all modes of this option.</p> <p>The assembler always starts in plain #SP mode (no offsets).</p> <p><i>See also internal symbols :SP and :SP1 and the simulated indexed modes ,ASP and ,LSP</i></p>
#SPCHECK	<p>#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</p>

	<p>stack situation (assuming correct placement of the relevant directives).</p> <p>The current difference between <code>:SP</code> and <code>:SPCHECK</code> is found in <code>:SPFREE</code> (e.g., use with <code>AIS # :SPFREE</code>)</p> <p>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.</p> <p>Hint: If you do not wish to use the <code>#SPAUTO</code> function for a particular section of code (or anywhere in your program) you can still temporarily place the <code>#SPAUTO</code> directive at the beginning of a code section to check, and the <code>#SPCHECK</code> 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.</p> <p><i>See also #SPAUTO</i></p>
<p>#x [expr]</p>	<p>When <code>#x</code> 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 <code>TSX</code>), or anytime the same constant needs to be added to a series of X-indexed instructions within a block of code.</p> <p><code>#PUSH</code> and <code>#PULL</code> will save/restore the current setting of this option.</p> <p>The assembler always starts in plain #x mode (no offsets).</p> <p><i>See also internal symbol <code>:x</code> and the simulated indexed mode <code>,AX</code></i></p>
<p>#UNDEF symbol [, symbol] *</p>	<p>Undefines one or more symbols. If a symbol is not currently defined, a warning will be issued (to protect from possible typing errors).</p> <p>Careless, or simply wrong use of this directive can lead to multiple side errors or warnings (please note this is a two-pass assembler).</p>

	<p>If you simply want to redefine the value of a symbol, prefer using the <code>SET</code> pseudo-op, rather than using <code>#UNDEF</code> followed by a repeated symbol definition.</p> <p><code>#UNDEF</code> can be used, for example, to completely remove unrelated or conflicting conditionals.</p>
#PAGE	<p>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.</p>
#PUSH	<p>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, HCSx, MMU, JUMP, CALL, S1, S2, SP1, SP, SPAUTO, X, TRACEx, MACRO, @MACRO, MCF, MACROSHOW, MACROHIDE, various SP-based offsets (eg., :AIS), :PSP, :PPC:, TRACE[ON/OFF], MLISTx and TABSIZE. Useful in included files that want to change any of these options without affecting parent files. See also #PULL</p>
#PULL	<p>Pulls from an internal stack the most recently pushed options. See also #PUSH</p>
#RAM	<p>Activation of the RAM segment. Default starting value is \$0080.</p>
#ROM	<p>Activation of the ROM segment. Default starting value is \$F600. This is the default segment if none is specified.</p>
#SEG_n	<p>Activation of the SEG_n segment (n is a number from 0 through 9). Default starting value for all ten segments is \$0000.</p>
#TABSIZE <i>n</i>	<p>Specifies the field width of tab stops used in the source file. Proper use of this directive ensures that the *.LST files generated by ASM8 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 for the source file(s) in which it is encountered.</p>
#TEMP [<i>expr</i>]	<p><code>#TEMP</code> simply assigns any value (possibly the result of a non-forward expression) to the internal general-purpose <code>:TEMP</code> variable. If no expression follows <code>#TEMP</code>, <code>:TEMP</code> is zeroed.</p> <p><code>:TEMP</code> 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 <code>:TEMP</code> always refers to the most recent <code>#TEMP</code> directive, so it cannot be used to look forward.</p> <p>Although <code>:TEMP</code> is a single variable, its use is transparent in relation to macros. In other words, changing <code>:TEMP</code> from</p>

	<p>within any macro does not affect the value of <code>:TEMP</code> outside all macros, or macros above the current level.</p> <p>Although macros inherit their initial value of <code>:TEMP</code> from their higher level (either a caller macro, or normal code), they do not affect their parent's <code>:TEMP</code> value, so you can use it without worrying about side effects from any intermediate macro calls.</p> <p><code>:TEMP</code> is also assigned <i>indirectly</i> when used as label with th any of the following directives/pseudo-ops: <code>NEXT</code>, <code>NEXP</code>, <code>SETN</code>, and <code>#AIS</code></p>
<p>#TRACEON #TRACEOFF</p>	<p>#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.)</p> <p>#TRACEOFF turns this option off making macros appear as a single line in the debugger. #TRACEOFF is the default state when assembly is started.</p>
<p>#VARIABLE <code>addr1</code> [<code>addr2</code>] #VARIABLE <code>#OFF#</code></p>	<p>Maps a location (or range, if <code>addr2</code> is also supplied) of variable allocation area (normally in RAM) as valid. Use multiple directives to specify additional ranges. Any <code>RMB</code> or <code>DS</code> definitions that fall (<i>fully or partially</i>) outside the given range(s) will produce a warning (if the <code>-O</code> option is enabled) for each such definition. <code>Addr1</code> and <code>addr2</code> may be specified in any order. The range defined will always be between the smaller and the higher values.</p> <p>The special keyword <code>#OFF#</code> removes all current definitions. See also #MEMORY</p>
#VECTORS	Activation of the VECTORS segment. Default starting value is <code>\$FFC0</code> .
#WARN	Turns warnings on. Equivalent to the <code>-WRN+</code> command line option. See also #NOWARN
#WARNING [<code>text</code>]	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 <code>\$2000</code> .
#XROM	Activation of the XROM segment. Default starting value is <code>\$EC00</code> .

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), and **(X)** for eXpanded. If no format modifier is used, (D) is assumed. 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
```

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. The potential uses of this capability are only limited by imagination.

An expression that cannot be evaluated (due to forward references or undefined symbols) will display as three question marks (???) when used 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:

- **::** 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: Beginning with v5.70 if any push instruction is followed by a label, that label will be SET to the current :: value (must be in #ExtraOn mode).*
- **:SP** returns the current offset of the `#SP` or `#SP1` directives. *This value is the basis for several other internal symbols.*
- **: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 `AIX` instructions. (Note: there are many ways to alter the contents of the `HX` 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 optimizing any `,SP` relative instruction to `,X` relative (by simply appending an `x` to `,SP` making it `,SPX` and using `TSX` 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 `AIX` instructions (like `:SPX`), it disregards possible following stack depth changes, unlike `:SPX`. (Note: there are many ways to alter the contents of the `HX` 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).
- **: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 and including the last stack-increasing `AIS` instruction (not the `#AIS` directive). For example, you may use it with a new [normally, stack-reducing] `AIS` instruction to free so many bytes of stack. `:AIS` is updated automatically after each stack-increasing `AIS` instruction, i.e., a negative `AIS` instruction with an actual instruction operand of `$80` to `$FF`, losing whatever previous value was in `:AIS`, and it can be used to free whatever stack bytes remain since the last stack-increasing `AIS` instruction (used like so: `AIS #: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` instruction.

- **: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.
- **:AB** returns the number of **A**ddress **B**ytes (useful for stack offsets). 2 for normal, 3 for MMU
- **:X** returns the current offset of the `#X` directive.
- **:YEAR** returns the year at assembly time (e.g., 2011) *Hint: Use `:YEAR\100` for two-digit year.*
- **:MONTH** returns the month at assembly time (e.g., 11)
- **:DATE** returns the date at assembly time (e.g., 22)
- **:CPU** returns a number representing the CPU type (6808 or 908)
- **:CRC** returns the current value of the running user CRC
- **:S19CRC** returns the current value of the running S19 CRC.
- **:PAGE_START** returns the page window's starting address.
- **:PAGE_END** returns the page window's ending address.
- **: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.

- **: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.
- **: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.
- **: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, after all, 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, after all, 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).

- **:LABEL** returns the length of the `~label~` placeholder's content used inside macros.
- **:TEMP** returns the current value of this 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.
- **:MLINENO** returns the current macro line number (only from within macros).
- **:ANRTS** returns the address of the most recent `RTS` instruction (i.e., always points back).
- **:ANRTC** returns the address of the most recent `RTC` 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.

Notes about `:SP` and `:SP1`:

:SP returns the current automatic SP offset (the same for both `#SP` and `#SP1` modes). It will NOT account for the extra 'plus one' of the `#SP1` mode, however. Use it to adjust the current SP offset manually, or to work with labels that are dependent on the current `#SP` or `#SP1` mode; for example, the current level labels while a one-based or zero-based offset is in effect. The local labels will still need to be in the same (zero or one) base, however. You can also use the simulated indexed mode **,LSP (Local SP)** to get the same effect.

:SP1, on the other hand, returns the current automatic SP offset just like `:SP` but which will be 'plus one' if we're currently in `#SP1` mode. This is always the true difference between automatic and actual stack offset, regardless of mode. You may also use the simulated indexed mode **,ASP (Absolute SP)** to get the same effect.

For example, use **,ASP** (or **-:SP1**) to cancel out all offsets for dealing with absolute numbers:

```

#sp1    SomeValue
...
?LocalA equ    1
pshx
cmp     1-:sp1,sp           ;absolute offset
cmp     ?LocalA-:sp1,sp     ;absolute offset
cmp     ?LocalA,asp        ;absolute offset
pulx

```

All three `CMP` instructions above will compare against the stacked value from the immediately preceding `PSHX` instruction, regardless if the directive earlier is `#SP` or `#SP1`, and regardless of the presence or value of the optional parameter *SomeValue*. It is equivalent to `CMP 1, sp` when `#SP` mode is off, and you can use it regardless of the current `#SP` mode and offset, and even if the `#SP` directive is never used, use it to lock the offsets (so that possible future `#SP`/`#SP1` automatic offsets in related code will not affect these lines).

This feature is most useful when a section of code is under `#SP` control (say, because most instructions refer to the parent routine's stack frame, and coding becomes simpler and more readable under `#SP` control) but you temporarily want to access local stack without any

automatic offsets. That way, you don't have to turn off #SP/#SP1 mode just for one instruction, or so.

To switch from zero-based label [or numeric] offsets to one-based (normal) stack offsets, without changing the current stack depth (automatic SP offset), you must do this:

```
#sp      :sp
```

Similarly, to switch back to zero-based offsets without changing the current automatic SP offset (current stack depth), you must do this:

```
#sp1     :sp
```

You can also use the :sp internal symbol to adjust X-indexed offsets after a TSX to refer to higher stack levels (say, a parent routine). For example:

```
MyOffset      equ      0           ;zero-based offset
               ais      #-1        ;allocate temp space
               ...
               tsx
               sta      MyOffset,x  ;save to local stack
               bsr      Sub
               ...
               #sp1     2           ;account for RTS (zero-based)

Sub            tsx
               lda      MyOffset,sp  ;gets A from parent stack
               lda      MyOffset+:sp,x ;(equivalent to above)
```

Examine the assembler listing on the following page to see the corresponding produced offsets for each case.

```

1      0000      ?      equ      0
2
3      F600      org      *
4
5                      ;#sp                      ; default mode, no offsets
6
7 F600:9EE6 01      [ 4]      lda      1-:sp,sp      ; SP/SP1 relative offset
8 F603:9EE6 01      [ 4]      lda      1,lsp         ; SP/SP1 relative offset
9 F606:9EE6 01      [ 4]      lda      1-:sp1,sp     ; absolute offset
10 F609:9EE6 01     [ 4]      lda      1,asp         ; absolute offset
11 F60C:9EE6 00     [ 4]      lda      ?,sp
12
13                      #sp1                      ; zero-based offset mode
14
15 F60F:9EE6 02     [ 4]      lda      1-:sp,sp     ; SP/SP1 relative offset
16 F612:9EE6 02     [ 4]      lda      1,lsp         ; SP/SP1 relative offset
17 F615:9EE6 01     [ 4]      lda      1-:sp1,sp    ; absolute offset
18 F618:9EE6 01     [ 4]      lda      1,asp         ; absolute offset
19 F61B:9EE6 01     [ 4]      lda      ?,sp
20
21                      #sp      10                ; one-based plus 10
22
23 F61E:9EE6 01     [ 4]      lda      1-:sp,sp     ; SP/SP1 relative offset
24 F621:9EE6 01     [ 4]      lda      1,lsp         ; SP/SP1 relative offset
25 F624:9EE6 01     [ 4]      lda      1-:sp1,sp    ; absolute offset
26 F627:9EE6 01     [ 4]      lda      1,asp         ; absolute offset
27 F62A:9EE6 0A     [ 4]      lda      ?,sp
28
29                      #sp1      :sp              ; zero-based plus :SP (10)
30
31 F62D:9EE6 02     [ 4]      lda      1-:sp,sp     ; SP/SP1 relative offset
32 F630:9EE6 02     [ 4]      lda      1,lsp         ; SP/SP1 relative offset
33 F633:9EE6 01     [ 4]      lda      1-:sp1,sp    ; absolute offset
34 F636:9EE6 01     [ 4]      lda      1,asp         ; absolute offset
35 F639:9EE6 0B     [ 4]      lda      ?,sp

```

Notes about :AIS:

:AIS returns the number of stack bytes still allocated since the most recent stack-increasing AIS instruction (normally useful only in #SP[AUTO] modes). Example usage:

```

#spauto                      ;auto mode and zero offset

push                          ;protect all registers

ais      #-5                  ;(negative AIS marks the current :SP)
...
ais      #2                    ;de-allocate some locals (no marking)
...
ais      #:ais                 ;deallocate however many bytes (3 in this case)
                                ;still left on stack since negative AIS

pull
rts

#sp                            ;cancel auto mode and offsets

```

Notes about #SPAUTO:

Besides providing a method for automatic adjustment of stack offsets that refer outside/within a certain block of code (so they remain 'apparently' constant), #SPAUTO can also be used to automatically de-allocate the correct number of stack bytes at the end of a block of code. This allows for easier and more accurate coding (since no mental accounting is required) and possible future changes in existing code do not require adjustment of any 'usually' affected instructions.

The following is an example of code that shows both of the above uses:

```

#spauto                      ;auto mode and zero offset

ldhx      Size,sp
pshhx

```

```

ldhx      ToAddress,sp      ;no need for: ToAddress+2,sp
pshhx
ldhx      FromAddress,sp   ;no need for: FromAddress+4,sp
pshhx
call      CopyMemory       ;call the action routine
ais       #:spfree         ;deallocate however many bytes
                               ;were allocated for parameters
...
#sp       ;cancel auto mode and offsets

```

Now, as an example, if you wanted to add yet another set of parameters to the CopyMemory routine (assuming the routine was modified to accept them) you would add some extra push instructions to the above code (shown in orange in the modified code below), but no other code or offsets would need adjusting. The same example becomes:

```

#spauto           ;auto mode and zero offset

ldhx      #SkipTo       ;new instruction added later
pshhx
ldhx      #SkipFrom     ;new instruction added later
pshhx
ldhx      Size,sp       ;new instruction added later
pshhx
ldhx      ToAddress,sp  ;no need for: ToAddress+2,sp
pshhx
ldhx      FromAddress,sp ;no need for: FromAddress+4,sp
pshhx
call      CopyMemory    ;call the action routine
ais       #:spfree      ;deallocate however many bytes
                               ;were allocated for parameters
...
#sp       ;cancel auto mode and offsets

```

To appreciate how #SPAUTO can help, look at the same changed code but without #SPAUTO, where all three SP indexed instructions plus the AIS instruction would have to have their offsets appended with an extra +4 (not to mention the original extra offset that must be in place as stack grows with each push). After a while this can get messy and error-prone.

```

ldhx      #SkipTo       ;new instruction added later
pshhx
ldhx      #SkipFrom     ;new instruction added later
pshhx
ldhx      Size+4,sp     ;new instruction added later
pshhx
ldhx      ToAddress+2+4,sp ;2 (original) +4 (for new)
pshhx
ldhx      FromAddress+4+4,sp ;4 (original) +4 (for new)
pshhx
call      CopyMemory    ;call the action routine
ais       #6+4         ;deallocate 6 (original)
                               ;+4 (for new)

```

With automatic SP offsets, you can't be sure at all times of the actual offset used (after all, the idea is to let the assembler decide automatically), so certain optimizations that depend on specific stack offsets may no longer work if stack ordering is later changed. For example:

```

pshhx           ;Place number to divide on stack

ldhx      #10
pula
div
psha
lda      2,sp

```

```

div                ;LSB
sta                2,sp
pulhx              ;deallocate however many bytes
                  ;were allocated for parameters

```

will work. But, if it were changed to:

```

pshhx              ;Place number to divide on stack
#spauto           ;auto mode and zero offset
psha              ;added later to protect A
ldhx              #10
pula              ;MSB (assuming it is at TOS)
div
psha              ;added later ...
lda                2,sp
div                ;LSB
sta                2,sp
pula              ;added later ...
...
#sp               ;cancel auto mode and offsets

```

it will no longer work because PULA/PSHA (shown in orange) always work on the top-most stack element (*#SPAUTO cannot alter those instructions*), which in this case is no longer the number we want to divide (as PSHA before LDHX – shown in green – has placed its own value on the top-of-stack, as we now decided to protect register A from destruction during the division operation) moving the MSB of the number we want to divide to the true offset 2, SP.

Here's then what to do to have the assembler use the optimization but only if the TOS (top-of-stack) has the number:

```

#spauto           ;auto mode and zero offset
?Number          pshhx      ;Place number to divide on stack
set              ::         ;Give name to just-stacked number
;;;;;;;;;;;;;;;;;; pshhx      ?Number    ;Equivalent to previous two lines combined (#ExtraOn required)
psha              ;added later to protect A
ldhx              #10
#iftos ?Number   pula
                  div        ;MSB (assuming it's at TOS)
                  psha
#else
lda                ?Number,sp
div                ;MSB
sta                ?Number,sp
#endif
lda                ?Number+1,sp
div                ;LSB
sta                ?Number+1,sp
pula              ;added later ...
...
ais                #:spfree  ;deallocate however many bytes
                  ;still allocated on stack
...

```

```
#sp ;cancel auto mode and offsets
```

Here, we use `#SPAUTO` outside the whole code block. We then assign a name to the stacked number (two methods shown, 2nd one commented out) using the `::` internal variable (which always – i.e., regardless of current stack depth and the `#SP/#SP1` setting of `#SPAUTO` mode – refers to the top-of-stack item, so it must be placed immediately following the stack operation – i.e., before another stack altering operation). Finally, using conditional assembly (`#if tos`), we test for the `?Number sp` offset having the effective value 1 (i.e., TOS), and if so, we can use the optimization, otherwise we code it as if it were any other stack offset. (This is particularly useful in general-purpose macros, where it's not possible to know in advance if the operand is going to be the TOS or not, but we want to have the best possible code generation for each case.) If the instruction pair `PSHA/PULA` (shown in green) is later removed, no changes are required to any of the remaining code. The `PULA/PSHA` optimization will automatically be activated, and due to `#SPAUTO`, all offsets will be adjusted accordingly.

The above example also introduces the use of the `::` internal symbol.

The special `::` internal symbol (a shortcut for the equivalent expression `1-:SP` for `#SP[AUTO]` modes, or `0-:SP` for `#SP1` mode) is a very useful dynamic symbol for quickly and easily (i.e., without any mental calculations) assigning labels to any stacked content, and at the exact time you push anything on the stack (no need to have your stack frame organized in advance, just define it as you use the stack). Because of the automatic SP adjustment while in the `#SPAUTO` mode, a symbol defined this way will always point to the correct location in the stack. No possibility of user error (assuming correct placement of `#SPAUTO` and/or related directives). *This symbol will always point to 1, SP if used outside of #SP[AUTO] modes (or 0, SP in #SP1 mode – which is practically useless), and a warning will be issued whenever you use a stack-related internal symbol outside of a relevant SP-adjusting mode.*

To use, always start a sub-routine with `#SPAUTO`. Any stack offsets 'above' the current routine (normally, that would be in the parent routine's stack) will be defined 'before' the `#SPAUTO` directive (not necessarily 'physically before' but 'logically before', i.e., based on a fixed starting offset value, usually the value one, and NOT the dynamic value of `:SP`), while stack contents within the routine will be defined 'after' the `#SPAUTO` directive using the `::` internal symbol as a base offset. An `#SP` directive after the end of the sub-routine turns all automatic SP adjustments off.

You may even redefine a stack-offset symbol (using the `SET` pseudo-instruction) and use it again with an updated stack location to prevent accidentally accessing the wrong stack item. For example, copying a parent routine's variable to local stack so that you may change the local copy without affecting the parent stack's variable is as simple as:

```
#spauto ;auto mode without any offsets
Parent pshhx ?Parm ;Place parm on stack and name it
...
```

```

?SP_before_call    equ      :sp          ;Mark SP offset before call
                   bsr      Child
                   ...
                   sthx     ?Parm,sp      ;Affects Parent's stack variable
                   ...
                   rtc

                   #spauto  2+?SP_before_call ;auto mode plus RTS and parent SP offsets

Child
                   ldhx     ?Parm,sp      ;Get parm from parent's stack
                   pshhx    ?Parm        ;Place parm on local stack and name it
                   ...          ;(Name "?Parm" now refers to Child's copy)
                   sthx     ?Parm,sp      ;Affects Child's stack variable
                   ...
                   rts

                   #sp          ;cancel auto mode and offsets

```

Once you see in practice how this (#SPAUTO and related language extensions) can help you maintain the correctness of all stack references in a program as you alter stack operations (e.g., add new push/pulls, or change the order of push/pulls – *you're still responsible for keeping the correct LIFO order yourself, however*), you may be surprised how you ever managed without it. Certainly, I was!

Several coding examples that collectively implement all the features described here can be found at <http://www.aspisys.com/code> followed by the link shown on that page, for the corresponding example. (To prevent auto-cloning of these pages, you may have to manually type the rest of the URL, as no direct links may be provided for each separate example.)

Notes about :x and #x mode.

:**x** will return the current automatic X offset. This allows you to get the true difference between automatic and actual X offset. You can also use the simulated indexed mode ,**AX** (**Absolute X**) to get the same effect.

As an example, you can use the #x mode to adjust X-indexed offsets after a TSX to refer to higher stack levels (e.g., parent routine). For example:

```

MyOffset           equ      0          ;zero-based offset
                   ais      #-1        ;allocate temp space
                   ...
                   tsx
                   sta      MyOffset,x ;save to local stack
                   bsr      Sub
                   ...
                   #sp1     2          ;account for RTS (zero-based)
                   #x      :sp

Sub
                   tsx
                   lda      MyOffset,sp ;gets A from parent stack
                   lda      MyOffset,x ;(equivalent to above)

```

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 you do not have to keep separate cycle counts for 9S08 and HC08 families. A third 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:

```

Delay10ms      #Cycles                ; reset cycles counter
               pshhx
               ldhx      #10*BUS_KHZ-?ExtraCycles/?LoopCycles
?ExtraCycles   equ       :cycles      ; grab counter (and reset)
?Delay.Loop    aix       #-1
               cphx      #0
               bne       ?Delay.Loop
?LoopCycles    equ       :cycles      ; grab counter (and reset)
               pulhx
               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 ASM8
; Input  : StackLo = StartAddress
;         : StackMd = EndAddress
;         : StackHi = Initial/Previous CRC
; Output : Stacked CRC updated
; Note(s): Call repeatedly for different address ranges, if skipping sections
; Call   :      ldhx      #CRC_SEED
;         :      pshhx
;         :      ldhx      #EndAddress
;         :      pshhx
;         :      ldhx      #StartAddress
;         :      pshhx
;         :      call     GetAsmCRC
;         :      ais      #4
;         :      pulhx
;
?
?StartAddress      set      1
?EndAddress        next     ?,2
?CRC               next     ?,2

#spauto           :ab           ;account for [RTS/RTC]

GetAsmCRC         push

?GetAsmCRC.Loop   ldhx      ?StartAddress,sp
                  cphx      ?EndAddress,sp
                  bhi      ?GetAsmCRC

                  lda      ,x
#ifdef COP        sta      COP           ;in case of many iterations
#endif
                  beq      ?GetAsmCRC.Next
                  cbeqa    #0xFF,?GetAsmCRC.Next

                  mul
                  add      ?CRC+1,sp
                  sta      ?CRC+1,sp
                  txa
                  tsx
                  adc      ?CRC,sp
                  sta      ?CRC,sp

                  ldhx      ?StartAddress,sp
                  lda      ,x
                  thx
                  mul
                  tsx
                  add      ?CRC,sp
                  sta      ?CRC,sp

?GetAsmCRC.Next   tsx
                  inc      ?StartAddress+1,sp
                  bne      ?GetAsmCRC.Loop
                  inc      ?StartAddress,sp
                  bra      ?GetAsmCRC.Loop

?GetAsmCRC        pull
                  rtc

#sp
```

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 ASM8

- 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 ; comments).
- All shaded areas are pertinent to the MMU enabled versions (Win32/Linux only).

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' comparison for the \$IF directive.
<>	'Not equal' comparison for the \$IF directive.
>=	'Greater than or equal' 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' 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 constants 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 from lower 16-bit word (unary): [\$1234 = \$34
]	Extract high 8 bits from lower 16-bit word (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 «<»)

ASM8 Extended Instruction Set

The instructions listed below are not actually new instructions, rather, internal macros that generate one or more HC08/9S08 CPU instructions. These instructions are only recognized if the extended instruction set option is enabled (-X+ command line option or #EXTRAON processing directive), and are used just like normal instructions, but NOT like user-defined macros.

Mnemonic/Syntax	Description
AAX	Add A to H:X Same as: PSHA/TXA/ADD 1, SP/TAX/THA/ADC #0/TAH/PULA (Last updated in v8.31 for one byte smaller size and fewer cycles)
ABS	Absolute value of A (note: value \$80 does not change) Same as: TSTA/BPL ?/NEGA/?
ADDHX #wordval	Add immediate value to HX (useful for table offset adjustment) Same as: PSHA/TXA/ADD #LSB/TAX/THA/ADC #MSB/TAH/PULA
CLRHX	Same as: CLRH/CLRX
CMPA operand	Same as: CMP operand
CMPX operand	Same as: CPX operand
DEX	Same as: DECX
INX	Same as: INCX
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)
JHCC addr16	Jump equivalent to BHCC (BHCS \$+5 followed by JMP addr16)
JHCS addr16	Jump equivalent to BHCS (BHCC \$+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)
JIH addr16	Jump equivalent to BIH (BIL \$+5 followed by JMP addr16)
JIL addr16	Jump equivalent to BIL (BIH \$+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)
JMC addr16	Jump equivalent to BMC (BMS \$+5 followed by JMP addr16)
JMI addr16	Jump equivalent to BMI (BPL \$+5 followed by JMP addr16)
JMS addr16	Jump equivalent to BMS (BMC \$+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)

LSLHX	Logical shift left of HX (useful for table offset adjustment) Same as: PSHH/LSLX/ROL 1, SP/PULH
NEGHX	Same as: PSHH/COM 1, SP/PULH/COMX/AIX #1
NEGXA	Same as: COMX/NEGA/BNE ?/INCX/?
OS <i>byteval</i>	Operating system call: SWI / DB <i>byteval</i>
OSW <i>wordval</i>	Operating system call: SWI / DW <i>wordval</i>
PSHCC	Same as: PSHA:2/ TPA/STA 2, SP/TAP/PULA
PSHHX	Same as: PSHX / PSHH
PSHXA	Same as: PSHA / PSHX
PULCC	Same as: PSHA / LDA 2, SP/TAP/PULA/AIS #1
PULHX	Same as: PULH / PULX
PULL	Same as: PULH / PULX / PULA
PULXA	Same as: PULX / PULA
PUSH	Same as: PSHA / PSHX / PSHH
RESET	Illegal instruction sequence to force MCU reset (Opcode used: \$9E9E)
SEXA	Sign-extend A to XA: CLR X / TSTA / BPL ? / COMX / ?
TAH	Same as: PSHA / PULH
THA	Same as: PSHH / PULA
THX	Same as: PSHH / PULX
TXH	Same as: PSHX / PULH
XGAX	Same as: PSHA / TXA / PULX
XGAH	Same as: PSHA / THA / PULH
XGHX	Same as: PSHH / TXH / PULX

ASM8-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.

All shaded areas are pertinent to the MMU enabled versions (Win32/Linux only).

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 24 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/SET 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**
Found 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: `$ABCH` and `$FFFFFF` 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.

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.

41. Invalid argument for DW or FDB directive

The value or expression supplied is not correct.

42. Invalid argument for FAR directive

The value or expression supplied is not correct.

43. Invalid argument for ORG directive

The value or expression supplied is not correct.

44. Invalid argument for RMB or DS directive

The value or expression supplied is not correct.

45. Invalid argument for END directive

The value or expression supplied is not correct.

46. MMU is disabled

The MMU extensions are disabled (-MMU- or #NOMMU in effect). The instructions CALL and RTC will produce error(s), as the target MCU may not support the MMU extensions (if the current MMU switch state was intentional).

WARNINGS

All shaded areas are pertinent to the MMU enabled versions (Win32/Linux only).

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

Automatic Direct Mode detection requires that 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.

2. **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).

3. **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).

4. **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 the list is separated by both commas and spaces. You must either remove the spaces or assemble with `#SPACESON` (or the `-SP+` option).

5. **Violation of MEMORY directive at address \$<VALUE>**

6. **Violation of VARIABLE directive at/near \$<VALUE>**

ASM8 has produced code and/or data that falls outside any address ranges defined via the `#MEMORY` or `#VARIABLE` directive. 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).

7. **EQU/EXP/SETs 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`.

8. **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.

9. **String is too long, only first 8 or 16 or 24 bits used**

8-bit instructions (such as `LDA`) cannot accept a constant string value of more than 8 bits. The longer string encountered is truncated before being used.

10. **S19 overlap at address \$<VALUE> [Linear: \$xxxxxx]**

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!

11. **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.

12. **No ending string delimiter found**

The last string quote is missing. ASM8 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).

13. **Operand is larger than 24 bits, using low 24-bits**

The operand is greater than 24 bits but the instruction can only accept a 24-bit operand. The lower 24-bit word was used.

14. **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.

15. **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.

16. **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.

17. **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. A `CALL` instruction, when used as a `JSR` (either directives `#NoMMU` and `#JUMP` or command-line options `-MMU-` and `-J+` are in effect) will NOT display this warning (if using the latest version and build). This is so you do not have to use `#OptRelOff/#OptRelOn` around all calls when not in MMU mode.

18. **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.

19. **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 is 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.

20. **No ORG (RAM:\$0080 ROM:\$F600 DATA:\$FE00 VECTORS:\$FFDE)**

ASM8 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.

21. 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, whether it's 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.

22. 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.

23. 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 ASM8 that supports additional directives your version doesn't understand.

24. USER: <USER TEXT>

This is a user generated warning via the #WARNING directive

25. 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.

26. 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.

27. Instruction BGND is only valid in BDM

This instruction is practically never used in a user program (except perhaps to force an illegal opcode exception). It's only good for debugging purposes in the special BGND mode of HCS08 CPUs only. Not applicable to the HC08 CPU. Available only in -HCS+ (HCSON) mode.

28. Attempt to JUMP to page. Use CALL instead.

You are using a Jxx instruction to send control to paged code (when in MMU mode, only). This will never work if jumping from one page to another, and it might work if jumping from non-page to page (but only if PPAGE is set correctly before the JUMP).

29. :<INTERNAL_SYMBOL> is useless outside of #SP[AUTO] mode(s)

You are using the shown stack related internal symbol (such as ::, :SP, :SPFREE, etc.) outside of #SP[AUTO] modes. The produced stack offset may be pointing to an incorrect stack location.

30. Stack structure requires 'AIS #-<value>'

Shown only via the #AIS directive. If a mismatch is found, the correct AIS instruction is shown. See the #AIS directive.

31. Invalid SP offset (<value>)

When the effective SP offset for any SP-indexed instruction is less than 1 (a condition normally not allowed except in extraordinary situations and only with interrupts disabled), this warning shows the effective SP offset you're attempting to use. If the action is intended (very unlikely), you can turn off the warning by enclosing the affected instruction(s) in `#NoWarn` / `#Warn` directives.

This warning is particularly useful with `::` defined labels in `#SP[AUTO]` modes, protecting you from using a symbol that refers to already released stack, which due to the automatic SP offset adjustment will point to a true offset of less than 1.

32. <"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).

ASM8'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

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 `$F600`).

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 of individually marking 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

ASM8 -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 * QE128: Target is MC9S08QE128
#Message * GP: Target is MC68HC908GP32
#Message *****
#Fatal Run ASM8 -Dx (where x is any of the above)
#endif
```

The command **ASM8 -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
ROM      def      $F800      ;Default ROM location

Start    ORG      ROM
         rsp      ;the program begins here
         ...      ;rest of program is left to imagination
         bra      *      ;the program ends here
```

will be assembled at \$F800 with the command `ASM8 SAMPLE` but you could also assemble with a command similar to this: `ASM8 SAMPLE -DROM:$8000` 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
ARRAYSIZE def      10      ;Default size for array

#if ARRAYSIZE < 2 ;check for minimum size allowed
    #Warning ARRAYSIZE ({ARRAYSIZE}) must be at least 2
ARRAYSIZE set      2      ;Minimum size for array
#endif

        #RAM

Status  rmb      ARRAYSIZE
Pointer rmb      ARRAYSIZE*2
        ...

        #ROM

Start   rsp      ;the program begins here
        ...      ;rest of program is left to imagination
        bra      *      ;the program ends here
```

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!

Calling ASM8 (DOS version) from PE's WinIDE

Here's how to setup the WinIDE to use ASM8.EXE instead of PE's assembler. This will let the IDE catch possible errors during assembly and take you to the problem source file and line.

Create a batch file, called ASM8.BAT, which contains the following lines:

```
@echo off
c:\utils\asm8.exe %1 %2 %3 %4 %5 %6 %7 %8 %9 -T+ >c:\temp\error.out
```

where `c:\utils` is the directory `ASM8.EXE` is located and `c:\temp` is a directory you want to use for sending the temporary file with errors.

In WinIDE, select "Environment -> Setup Environment -> Assembler/Compiler -> EXE Path"

Put the path of the ASM8.BAT file you created earlier.

In the box Type, select "Other Assembler/Compiler"

In Options, type: `%FILE%`

Select the checkboxes for "Wait for compiler to finish", "Save files before assembling", and "Recover Error from compiler" and deselect the remaining ones.

The Error Filename should be `c:\temp\error.out` (or whatever filename you actually used in your ASM8.BAT above)

The Error Format should be "Borland Compatible"

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. 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      rsp                ;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 refers primarily to the DOS/Win version. All shaded areas are pertinent to the MMU enabled versions (Win32/Linux only).

Note: Due to the absolute memory restrictions imposed by pure DOS architecture, the DOS version of the assembler will no longer be expanded with any new features as this would limit the maximum work space (e.g., number of possible label definitions, etc.) Should any bugs be found in existing features, these will be corrected, however.

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 v1.20, 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 asm8.cfg in the same directory as the asm8 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 asm8.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 asm8 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 asm8 binary is, and anytime a local asm8.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, e.g., the various MMU extensions shown with shaded text throughout this document, are available only in the Win32/Linux versions.