ASM8 68HC08/9S08 Macro Assembler User's Manual

ASM8 - Copyright (c) 2001-2022 by Tony Papadimitriou (email: tonyp@acm.org)

Latest Manual Update: February 13, 2022 for ASM8 v12.30

ASM8 is a two-pass absolute macro cross-assembler for the 68HC08 or HCS08 or 9S08 MCU by NXP (originally by Motorola, and later by Freescale).

This tool has been evolving for almost 20 years with new features, and occasional bug fixes, with very few and trivial backward compatibility issues. It is mature enough to cope with the most demanding applications both in terms of size and coding complexity.

There are many *sometimes unique* features that make writing clean, self-adjusting source code easy. Those make working with assembly language so much more fun *for me, at least*, than with certain higher level languages.

Most higher level language tools usually come with an ecosystem, a set of built-in libraries for the most mundane tasks a programmer needs. This is something most assembler writers unfortunately do not bother to include with their tools making it harder for newcomers to start using the language, successfully. And, eventually, they escape frustrated into other languages.

Although there's a lot of code floating on the Net, this is often unreliable, untested, incomplete (fails to assemble out of the box), of little or no practical value, badly formatted and/or unreadable, and written for diverse *sometimes unspecified* assemblers whose details are not always clear making porting to your own assembler difficult and error prone. Most importantly, all this code, collectively, lacks consistency in coding style, naming and calling conventions, making it that much harder to integrate into your own application without significant rewrites.

I have been trying to change this trend. ASM8 comes with a significant royalty free code base, most of which has been in use in real life applications and proven over time. These libraries *(found under the `code` subdirectory)* in ASM8 distribution (**freeasm8.zip**) are *wherever possible* re-entrant, self-contained, and include the following:

- A rich set of general-purpose macros
- String functions
- Copy functions
- Internal Flash functions (*for the most common MCU variants*)
- Self-timed delays *(based on user defined bus frequency)*
- A special math library that can be used *via a single `Eval` macro* with the same ease as when writing math expressions in higher level languages *(taking care of all lower level details such as operator precedence, and mixing variables of different sizes)*
- A general-purpose `print` macro that can print a mix of constants, constant or variable strings, math expressions, and pointed strings. Using unified I/O, you only have to define `putc` macro to tell it where you want its output to go.
- A standalone tiny bootloader that accepts S19 records directly from an SCI (RS-232 or USB) attached terminal, such as PuTTY.

The above can be used as is, or become inspiration for even better ones.

My plans are to continually enrich this library not only with my own code but also code donated by others, *hopefully*. I urge anyone interested to donate any non-proprietary, well crafted, and tested libraries and/or applications. Please contact me if you have something to offer.

I happen to use ASM8 on a daily basis on a great number of applications and libraries written 100% in assembly language (*well over one million lines of code by now*).

Three versions are currently available:

- 32-bit Windows *(also runs under 64-bit Windows)*
- i386 Linux
- DOS *with the GO32V2 memory manager extension built-in*. It has also been tested under **DOSBox v0.73-2** and **DOSBox-X 0.83.9** and performs without problems.

See Linux/Win32 version addendum for behavioral differences related to the OS.

Version History

```
+------+-----------------------------------------------------------+
|12.30 |Added option -? for interactive definition of no-operand DEF |
|12.20 |Added target address for BRCLR/BRSET instructions in listing |
|12.12 |Removed redundant errors/warnings w/ inherent B[R]SET/B[R]CLR|
|      |Added LSRHX extra (for dividing HX by two)                   |
|      |Removed 'UNKNOWN' from unknown-mode instruction errors       |
|12.11 |BugFix: #SEGMENT with ORG parameter would fail under SpacesOn|
|12.10 |Added -H option & #[NO]HINTS to show/hide hints (default ON) |
|12.01 |Minor internal bug fix in relation to PIN pseudo-op          |
|12.00 |Major new feature release!                                   |
```

```
|       |Added PIN pseudo-op to define port pins and zero-page flags  |
|       |Added :PIN internal symbol to return last PIN used status    |
|       |Added BCLR/BSET/BRCLR/BRSET recognition of PIN defined labels|
|       |BugFix: Larger than 16-bit values are now #EXPORTed correctly|
|       |BugFix: (since v8.90) size would update in false conditionals|
|11.95  |Changed some messages from stderr to stdout to be captured   |
|       |by redirection                                               |
|11.90  |Allow expression as 1st parm to #EXPORT when alias follows   |
|11.80  |Added optional 2nd parameter to #EXPORT to generate alias    |
|11.70  |Changed "Shorter form wasn't used" from warning to hint      |
|       |Changed "Direct mode wasn't used" from warning to hint       |
|11.60  |Added -FN non-saveable option to hide line numbers in listing|
|11.55  |Allowed && in #IF[N]DEF expressions                          |
|11.51  |BugFix: #IF[N]DEF ignores a plain ! (NOT) in expressions     |
|11.50  |#IF[N]DEF treats leading optional ! in expressions as NOT    |
|11.42  |BugFix: Macro parameters no longer expand @@ in labels       |
|11.41  |BugFix: ELSE checks for !IF as prefix instead of !IF exact   |
|11.40  |-Dxxx will update same symbol from a previous -Dxxx option   |
|11.31  |BugFix: 'Attempting global' would incorrectly flag some cases|
|11.30  |#IFTOS expression may be followed by ,index, e.g.: var@@,sp  |
|11.20  |#!MEMORY/#!VARIABLE hides warning on undefined expression    |
|11.12  |Improved ALIGN bugfix about honoring -F= command line option |
|11.11  |BugFix: ALIGN did not honor -F= command line option          |
|11.10  |A colon beginning command-line path will be relative to root |
|11.01  |BugFix: Segment ORG now shows errors in pass one like ORG.   |
|11.00  |Added :VERSION internal symbol as integer with two decimals. |
|       |Directives #S19WRITE, #HINT, #MESSAGE, #WARNING, #ERROR, and |
|       |#FATAL no longer trim the string to write.                   |
|       |Each segment now has its own ORG offset (possible minor      |
|       |backward compatibility issues).                              |
|       |BugFix: ORG with optional S19 expression did not always work |
|10.00  |Improved S19 overlaps in MMU mode                            |
|       |Added {...} format modifier (L) for hex8 without $ prefix    |
|       |Added {...} format modifier (W) for hex4 without $ prefix    |
|       |Added {...} format modifier (B) for hex2 without $ prefix    |
| 9.99  |All segments can now be followed by an ORG expression        |
|       |Added silenced error expression evaluation in ... comments   |
|       |Changed purely informative harmless warnings to hints        |
| 9.98  |Added warning under MMU on file-local CALL or global JSR/BSR |
|       |Added -G, #GCALL, and #LCALL to enable/disable this warning  |
| 9.97  |Added option -B[asename] to give a different output basename |
| 9.95  |Improvement: Label may be followed by comment without spaces |
| 9.94  |Added :MAXTEMP, :MAXTEMP1, :MAXTEMP2 internal variables      |
| 9.93  |Undefined macro call starting with ! is ignored without error|
| 9.92  |Added extra mnemonics `ASLH` (alias for `LSLH`), and `ASRH`  |
| 9.91  |OR condition separator can be either alt-179/0166 or '||'    |
|       |Made number of warnings and errors a long for huge numbers   |
| 9.90  |Added :CCYCLES to return current cycles without zeroing them |
|       |Changed hint 'Found [re]definition of' to '[Re]defined'      |
| 9.89  |BugFix: procname was affected even inside false conditional  |
|       |Show 'Found target address' hint macro name with -FX option  |
| 9.88  |Default: "Free private/commercial use license (not resale)"  |
|       |BugFix: Changed output to stdout for #ERROR and #WARNING     |
|       |Changed max command-line -F option targets from 1 to 100     |
| 9.87  |Improved NoICE map files and also removed proc local symbols |
|       |DEF without any operand now gives a more informative error   |
|       |Hint ... shows always and if prefixed with ! only when -D... |
|       |BugFix: Spaces between #EXPORT arguments were not trimmed    |
| 9.86  |BugFix: Correct implementation of empty string FCS case      |
| 9.85  |BugFix: Correct MMU implementation of OS8 extra instruction  |
|       |Warning 'P&E MAP files may not support more than 127 files'  |
|       |Added extra mnemonic TSTH (test H) as PSHH/TST 1,SP/PULH     |
|       |Added ... to mark incomplete sections with optional message  |
|       |BugFix: Define labels next to MTOP, MDO, MLOOP, and MSUSPEND |
|       |'Invalid SP offset' is a warning for zero but error for less |
|       |Remove all macro and proc local symbols from P&E map files   |
| 9.84  |#HomeDir now also tries _ASM_ and _FOSSIL_ before failing    |
|       |BugFix: Failure to #Include a file keeps counter unchanged   |
| 9.83  |CreateMap -MTA does not export byte size for missing labels  |
|       |Added option -WW to save configuration with license removed  |
|       |Optimized -U option processing                               |
|       |Allowed FCS to be an empty string as it creates trailing zero|
| 9.82  |Evaluate { ... } expressions w/out errors for informational  |
|       |directives inside macros                                     |
| 9.81  |Added OS8 extra instruction which does SWI followed by FAR   |
|       |#![M]EXPORT does not give warning on undefined label         |
| 9.80  |BugFix: #HOMEDIR first changes to the current file's path    |
|       |Improvement: #INCLUDE displays normalized path with -FI hint |
|       |Improvement: Added { ... } expressions in #INCLUDE paths     |
|       |Improvement: MTRIM error now displays actual problem text    |
|       |Improvement: Added AIS and AIX out-of-range warnings         |
```

```
| 9.79 |Added #TEMP1 :TEMP1 #TEMP2 :TEMP2 similar to #TEMP :TEMP      |
|      |#!DROP prevents 'macro not found' warning                       |
| 9.78 |Enhancement: !MRESUME does not give error                       |
|      |Enhancement: IF[N]DEF ORs alt-179/0166 separated conditions     |
|      |Added value display in 'Repeater out of range' message          |
|      |Added #!UNDEF to prevent 'symbol not found' warning             |
|      |Enhancement: EXIT accepts alt-179/0166 separated conditions     |
|      |Changed built-in defaults to -T+ and -C+                        |
|      |Added :ERRORS and :WARNINGS internal variables                  |
| 9.77 |BugFix: In -X+ mode, ignore * after a push instruction          |
|      |If '*' in FindSymbol target can be any substring of symbol      |
|      |BugFix: Error 'invalid binary number' once for same number      |
| 9.76 |MAP files when ?A or ?F root is present uses relative paths     |
|      |Absolute path is used when -U option is given                   |
|      |BugFix: RMB first defines label; allows its use in expression|
|      |Added -FI debugging option to show included filenames           |
|      |Altered the default segment addresses for more practical use    |
| 9.75 |#FATAL no longer aborts all assemblies, only the current one    |
|      |Allowed Escape to break out of a long repeater loop             |
|      |BugFix: Added recognition of \, as alias to • in macro calls    |
| 9.74 |Allowed !RMB or !DS to prevent warning about overlap            |
|      |Added #S19WRITE directive that writes messages to S19 file      |
|      |Added #S19RESET directive to forget all used S19 addr ranges    |
| 9.73 |Added -FX non-saveable option to enable macro line # display    |
|      |in ERROR, WARNING, MESSAGE and HINT directives (default: Off)|
|      |-F: option now also displays the related value                  |
|      |-F: option too now honors the newly added -FX option            |
|      |Help screen shows default -I option as DEFAULT_INCLUDE_PATH     |
| 9.72 |Added -LLn option to use a non-default label size up to         |
|      |MAX_LABEL_LENGTH. This option is global and cannot be saved.    |
|      |Added #MAXLABEL directive to specify new max label length.      |
|      |Added :MAXLABEL internal symbol to get the max label length.    |
|      |#PUSH/#PULL now save/restore the value of :MAXLABEL             |
|      |#MESSAGE and #HINT now show the macro line when inside macros|
| 9.71 |REMACRO now keeps previous definition even of special ? macro|
|      |Improved error precision by referring to exact directive name|
|      |Updated help screen to add TPX and TXP internal macros          |
| 9.70 |Added #ENDP, ENDP & :MAXPROC to allow nested proc definitions|
|      |BugFix: Define ENDP and ENDM labels before processing the       |
|      |actual directive so that they have local scope.                 |
|      |Added -F= alias to -F: for FindSymbol operation                 |
|      |-FQ no longer masks -F: operation                               |
|      |Changed max command-line -D symbol definitions from 10 to 100|
|      |Made 'Assembled' line formatting uniform in all cases           |
|      |Added TPX and TXP internal macros for TPA/TAP with X register|
|      |Augmented -F: option when numeric to find 1st use of address    |
| 9.69 |BugFix: #IFDEF processing always false on empty symbol          |
|      |BugFix: Added error if expression operand is empty string       |
|      |Added 'Elapsed time' display next to 'files processed'          |
|      |Added CALL, JSR, and BSR max stack depth checking               |
|      |Shortened warning message: Missing first operand in ...         |
|      |Added auto-fitting of 'Assembled' message to screen width       |
|      |Added :WIDTH internal symbol to return the width of screen      |
|      |Added warning 'Redundant AIS with zero operand'                 |
|      |Added -FH option to force hidden macros in listings             |
| 9.68 |Updated help screen to include internal symbols                 |
|      |Improved long name matching                                     |
|      |Made the default -I option equal to ?2;?1;?a;?f                 |
|      |BugFix: v9.65 case-insensitive compare in Windows               |
|      |Changed default tabsize to 10 to match proposed coding style    |
|      |BugFix: Only MSET has a #charset special merge option           |
|      |BugFix: Use platform specific path delimiter in -I option       |
|      |-FQ+ now also inhibits #HINT directive output                   |
|      |Added default include path in -I option if * is used empty      |
|      |Added -FE option to convert warnings to errors                  |
|      |Changed default -LC option to No                                |
| 9.67 |Added command option -F:symbol to find symbol (re)definitions|
|      |Added #ELSE IFxxx capability                                    |
|      |Added command option -FW to make free (non-error) warnings      |
|      |Added #HINT directive (like #MESSAGE but non-maskable)          |
|      |BugFix: Honored -T option for #MESSAGE directives also          |
|      |BugFix: -MMU option command-line processing                     |
| 9.66 |BugFix: Some macro processing occurred even inside false IFs    |
| 9.65 |BugFix: v9.63 path matching was wrong for many cases            |
| 9.63 |Hide Fossil (?F)/Asm (?A) root dir from listings                |
|      |Added better examples to -I option in help screen               |
| 9.62 |Added O (offset) and M (macro) indicator in listing             |
|      |Added dual address display in listing (actual and offset)       |
|      |Added ORG second parameter to allow separation of S19 address|
|      |Added :OFFSET internal symbol to return current offset value    |
|      |Added -FL non-saveable option to ignore #LISTOFF/#NOLIST        |
```

```
|      |Added -FM non-saveable option to ignore #MAPOFF            |
|      |Added saving/restoring of :OFFSET with #PUSH/#PULL         |
|      |Added ORG missing 1st parm (*-:offset) when 2nd is present |
|      |Allowed return to no offset current location by just "ORG ," |
|      |Added -F2 non-saveable option to force P&E 16-bit map      |
|      |Minor correction in help screen                            |
|      |BugFix: #USES now ignores filename case under Windows      |
|      |BugFix: Checking ?F and ?A now uses OR logic instead of AND |
|      |BugFix: Added ChDir(GetPath(path)) in List() for Linux case |
| 9.61 |Enhancement: -I option ?A matches special _asm_ file root  |
| 9.60 |Improved help screen for better fit in console window      |
| 9.58 |Added -FQ option to control display of line number display |
|      |during assembly. Useful for IDE use.                       |
|      |BugFix: -E option without +/- now works again              |
|      |Added extra mnemonics ROLH, RORH, LSLH, and LSRH           |
|      |Enhancement: -I option ?F matches FOSSIL root              |
| 9.57 |Internal changes only                                      |
| 9.56 |Enhancement: ?1 in -I option means main file path          |
|      |             ?2 in -I option means parent file path        |
| 9.55 |Enhancement: #MEXPORT now uses remacro instead of macro in |
|      |case of multiple exports of the same macro name.           |
| 9.52 |Enhancement: !BGND does not give warning (BGND does)       |
| 9.51 |BugFix: PUSH opsize presence suppressed redefinition warning |
| 9.50 |New INCLUDE search order: as is, parent path, include path |
|      |BugFix: IFDEF, IFZ, IF now use PosStr() instead of Pos()   |
|      |Added multiple -I option directories separated by semi-colon |
|      |Allowed user editable -I dir list when ? is given as parm  |
|      |-I '*' is a placeholder for current directory definition   |
|      |#!IF/#!IFZ/#!IFNZ false w/out warning on undefined expression|
| 9.45 |Added ExpToStr formats: Fn (space-fill) and Zn (zero-fill) |
|      |Added extra mnemonics INCH (increment H) & DECH (decrement H)|
| 9.43 |Corrected multi-operand pseudo-instructions size calculation |
|      |Added filename & linenumber display when canceling with ESC |
|      |Added :HOUR, :MIN, :SEC for clock time                     |
| 9.42 |Internal changes only                                      |
| 9.41 |BugFix: ::LABEL does not give error in pass one            |
|      |Improved NEXT when no label on left to increment placeholder |
| 9.40 |Made #SIZE second parm optional with :PC-LABEL as default  |
| 9.37 |BugFix: ~n[charset]index~ can contain [ and ]              |
|      |Charset can now also be delimited by " or ` quotes         |
| 9.36 |BugFix: Added "if MacroRecording then exit" to IF/ELSE/ENDIF |
| 9.35 |#IFNUM now detects bin & hex as numbers                    |
| 9.33 |Added ~text,~ and ~,text~ placeholders.                    |
| 9.32 |BugFix: Added missing size from EXP file symbols           |
|      |"Macro already defined" message now shows only in pass one |
|      |Added "Non-proc symbol redefinition" warning for PSHx      |
| 9.31 |BugFix in -MTA format when label is blank, size was misspaced|
| 9.30 |Added size info in -LSS and -MTA output formats            |
| 9.25 |Internal changes only                                      |
| 9.23 |BugFix: Added MSTOP 'USER: ' msg. Removed MERROR ExpToStr() |
| 9.21 |BugFix relating to $$$ internal conversion in macros       |
| 9.20 |BugFix: MergeParms now uses LastMacroParm                  |
| 9.18 |BugFix: v9.05+ did not allow ',' in EQU etc.               |
| 9.17 |Internal changes only                                      |
| 9.15 |MSET # now skips strings and allows quotes & parens in set |
| 9.10 |Allowed embedded { ... } expressions in macro calls        |
| 9.05 |Added ,size option next to EQU/SET/DEF                     |
|      |Added MSTOP in-macro parameter to display #Error           |
|      |Added #SPA alias for #SPAUTO directive                     |
| 9.02 |BugFix: ~n[set]index~ not working for all occurrences      |
|      |Added ~n'set'index~ alternative                            |
|      |Added ,size option as 'PSHx NAME,SIZE'                     |
| 9.00 |BugFix: > and < over 32-bit now do not wrap-around         |
|      |Added ~n[set]index~ macro parameter placeholder            |
|      |Added MTRIM macro command                                  |
| 8.90 |Added #SIZE LABEL,SIZE directive & ::label internal symbol |
|      |Added MSET # option to unite all parms into one            |
|      |Added :SPMAX, #SPAUTO, depth option                        |
|      |Added MSET #'charset' option                               |
|      |Made :SPMAX show maximum stack depth since last :SPLIMIT set |
|      |Added :SPLIMIT to return the currently effective stack limit |
|      |Added :NN internal variable to return total macro parm count |
|      |Minor speedup when using -Q+ option                        |
| 8.80 |Added placeholder ~[n.p]~ for smart part extraction        |
| 8.70 |#PROC resets ~procname~ to null                            |
|      |#AIS without argument resets :AIS counter to zero          |
| 8.60 |Added MDEL in macros                                       |
| 8.57 |Added MOV DIR,X+ & MOV X+,DIR instruction adjustment of :TSX |
| 8.55 |Added :N internal variable (for number of macro parameters) |
| 8.50 |BugFix: #MEMORY and #VARIABLE #OFF# did not work correctly |
|      |Added "RMB overlap" warning                                |
```

```
| 8.40 |BugFix: Symbols over 19 chars gave no error on redefinition  |
| 8.35 |Added ~filename~ ~basename~ ~path~ and ~#text~ placeholders   |
|      |Fixed bug with range in listing                               |
|      |Added #OFF# parameter to #MEMORY and #VARIABLE directives     |
|      |Added ProcName assignment, and ~procname~ placeholder(s)      |
| 8.31 |Improved code for the AAX internal macro                      |
| 8.30 |Allowed ALIGN to align only label to the left (if present)    |
|      |Added internal symbols to get current value of each segment   |
|      |Fixed bug with labeled ALIGN with zero                        |
| 8.21 |Fixed #MEXPORT by adding a new FindMacro in ASMACROS.PAS      |
| 8.20 |Added #REMACRO to allow renaming to an existing macro name    |
|      |Added ~self~ and ~text~ macro placeholders                    |
| 8.15 |Added MSUSPEND and MRESUME and fixed :MINDEX recursion        |
|      |MSTOP (with optional parm #ALL#) stops suspended macros       |
| 8.11 |Fixed "Used Range" percentage in case of #MEMORY use          |
|      |Allowed #EXPORT to come before the actual label definitions   |
| 8.10 |Fixed ALIGN with zero bug                                     |
| 8.09 |Added macro parm delimiter #PUSH/#PULL protection             |
| 8.08 |Added REMACRO capability                                      |
| 8.07 |Added MDO optional initializer expression                     |
|      |Added :CPU internal symbol                                    |
|      |Added #IFNUM and #IFNONUM to test a parm for being numeric    |
| 8.05 |Added MSTOP                                                   |
|      |Fixed long-standing bug #IFMDEF inside a macro                |
|      |Allowed first source line to have symbol also used by -D      |
| 8.04 |@macro or %macro can now start at column one                  |
| 8.03 |Fixed long-standing bug with #MCF[2] no-@ call               |
| 8.02 |Added #MCF2 mode                                              |
| 8.01 |First macro level inherits :TEMP from main                    |
| 8.00 |Added ;; comment removal from macros                          |
|      |Added ~macro~ and ~00~ placeholders                           |
|      |Added -S9 command-line option to disable S9 record            |
|      |Added #EXPORT directive                                       |
|      |Added #MEXPORT directive (for exporting macros)               |
|      |Changed from 'SET' to 'set' in EXP files                      |
|      |Fixed #MEXPORT to produce raw (fully unprocessed) macro lines|
|      |#MEXPORT uses preserve defined macro name case                |
|      |Added END INCLUDE message in #EXIT directive                  |
|      |Fixed #MEXPORT from within macro when -EXP-                   |
|      |Fixed some memory leaks related to macro use                  |
|      |Added :TSX internal variable                                  |
|      |Added :TEMP as possible label in NEXT, NEXP, SETN [#AIS]      |
|      |Added ALIGN pseudo-op                                         |
| 7.90 |Added :LINENO and :MLINENO internal variables                 |
| 7.85 |Added recognition of * and ?* wildcards to #DROP directive    |
|      |Fixed MREQ with message to show correct message               |
| 7.83 |Added :LABEL to return length of ~label~                      |
| 7.82 |Added -FD command-line option for fixed date in internal vars|
| 7.81 |Fixed #EXIT to restore conditional file level                 |
| 7.80 |Added #RENAME to rename a macro                               |
| 7.75 |Allowed @@ to #@Macro #Macro and #MCF for nested calls        |
| 7.72 |Allowed label next to MEXIT, etc.                             |
| 7.71 |Internal changes only                                         |
| 7.70 |Added #TEMP and :TEMP                                         |
|      |Non-macro :TEMP keeps its pre-macro value                     |
|      |Added == for IFPARM case-sensitive comparison                 |
| 7.60 |Added MDO MLOOP :MLOOP :MEXIT MTOP/MLOOP limits               |
| 7.50 |Added :ANRTS & :ANRTC internal variables                      |
|      |Added #EXIT directive to stop #INCLUDE file at that point     |
|      |Added #USES#USING directives to #INCLUDE if not-yet included |
| 7.40 |Added MERROR (combines #Error with mexit) for macros          |
|      |Added MSTR to turn non-string macro parm into string          |
| 7.30 |Added #IFB/#IFNB aliases for #IFNOPARM/#IFPARM, respectively |
|      |Added MREQ for required macro parameters                      |
| 7.21 |Internal changes only                                         |
| 7.20 |Added DEF which does EQU if not already defined               |
|      |BugFix: Command-line symbol definitions re-set for PASS2      |
| 7.11 |Internal changes only                                         |
| 7.10 |Added :PC and :PPC (#PPC)                                     |
| 7.02 |Added :OCYCLES to return the previous (Old) #Cycles           |
|      |MSWAP now silently ignores swapping with self                 |
| 7.01 |BugFix: Corrected cleanup of file-local macros on file change|
| 7.00 |Added recursive macros and #MLimit to set max depth           |
|      |Added MSET to change the text of a macro parm                 |
|      |:LOOP is reset when calling self with @@                      |
|      |Added MDEF (similar to MSET but only #IFNOPARM)               |
|      |Added MSWAP (swaps any two macro parameters)                  |
| 6.90 |Added #IFSTR/#IFNOSTR conditional directives                  |
| 6.80 |Allowed upto 125 total INCLUDE files                          |
|      |Added #PROC, PROC and :PROC for local labels                  |
|      |Allowed { ... } expression in labels even outside of macros   |
```

```
|      |Removed @@ local labels from SYM file                              |
|      |Added "Number of [#]PROCs used" message in LST                     |
| 6.70 |Internal changes only                                              |
| 6.61 |Ability to hide all macros macros & their definitions              |
| 6.60 |Added ability to hide all macro keywords                           |
|      |{ ... } expressions can now be nested                              |
| 6.50 |Internal changes only                                              |
| 6.45 |Added :DOW internal variable (0=Sunday)                            |
| 6.45 |Added MTOP                                                         |
| 6.42 |Did not allow negatives in ReadParm and ~n.s.l~                    |
| 6.41 |Length :0 now works correctly                                      |
| 6.40 |Added "Line may have been truncated" warning                       |
| 6.30 |Added :0 to :9 for getting length of macro parm                    |
| 6.21 |BugFix: #HOMEDIR not saving home between passes                    |
|      |Added #HOMEDIR without parms to restore original home              |
| 6.20 |Do not show #ENDIFs with -LC-                                      |
| 6.10 |Improvement: No trailing commas when reading macro lines           |
|      |Added: ~@~ and ~@@~ macro placeholders                             |
| 6.03 |BugFix: Improved v6.02 EQU & SET BugFix                            |
| 6.02 |BugFix regarding EQU & SET when used with repeaters                |
| 6.01 |Internal changes only                                              |
| 6.00 |Allowed { ... } expression> anywhere in macro text                 |
|      |BugFix: Other macro called right after #DROP                       |
|      |Added reset macro counters capability by calling %macro            |
| 5.90 |Added :INDEX for use from inside macros                            |
| 5.85 |:MACROLOOP -> :MACRONEST, added :MACROLOOP                         |
| 5.80 |Added :TOTALMACROCALLS, :MACROLOOP, :MACROINDEX                    |
| 5.71 |"Bug" fix. spxSP no longer updated with #SPAUTO                    |
| 5.70 |Added implicit "label set ::" next to any push instruction         |
|      |BugFix for space parm delimiter self-replacing                     |
| 5.60 |Added #Cycles to (re)set :CYCLES to any value                      |
| 5.55 |Added \, and • to re-use last macro delimiter                      |
| 5.51 |Added 'Nested macro definition not allowed'                        |
| 5.50 |Internal changes only                                              |
| 5.40 |Added #MCF (Macros Come First) & MacroFirst                        |
| 5.32 |Fixed misleading MMU "Attempt to jump to page"                     |
| 5.31 |Bug fix for ~#~ processing                                         |
| 5.30 |Added ~n,~ and ~,n~ macro placeholders                             |
|      |#PARMS SPACE makes separator a space                               |
| 5.25 |Added #IFSPAUTO conditional directive                              |
| 5.21 |Disallowed comments in #IFPARM/#IFNOPARM                           |
| 5.20 |Added ~label~ macro placeholder                                    |
| 5.12 |Fixed recognition of comma in non-@ macros                         |
| 5.11 |Internal changes only                                              |
| 5.10 |Added #IFPARM with comma [,] for comparing two strings             |
| 5.02 |Internal changes only                                              |
| 5.01 |Bug fix in ',SPX' case with zero :SP                               |
| 5.00 |Macro calling without the @ symbol >MacroAssume                    |
|      |Added warning "Macro repeater ignored"                             |
|      |Replaced #NOMACRO with #@MACRO                                     |
| 4.90 |Internal changes only                                              |
| 4.81 |Internal changes only                                              |
| 4.80 |Automatically drop special ad-hoc '?' macro                        |
| 4.70 |Added mapping for macro lines and #TRACEON/#TRACEOFF               |
| 4.60 |Added optional macro parm for single call                          |
| 4.52 |Fixed nested #IF(s) in macros not counted right                    |
| 4.51 |Internal changes only                                              |
| 4.50 |Internal changes only                                              |
| 4.40 |Internal changes only                                              |
| 4.30 |Made IFPARM/IFNOPARM available outside MACROS                      |
| 4.20 |Added IFMDEF/IFNOMDEF for check macro presence                     |
| 4.10 |Added #MLISTON/#MLISTOFF & #MLIST/#NOMLIST                         |
| 4.01 |Fixed local macro recognition                                      |
| 4.00 |Added MEXIT                                                        |
| 4.00B|Added macro support (BETA due to significant code changes)         |
|      |Added [n] display for macro errors/warnings                        |
|      |Added IFPARM/IFNOPARM, #DROP, #PARMS                               |
| 3.50 |Made Assembled display not truncate long names                     |
| 3.40 |-X option (dis-)allows simulated index modes                       |
|      |Added warning 'Invalid SP offset'                                  |
| 3.33 |Added AIX instruction adjustment of TSX-based SP                   |
| 3.30 |Added ,SPX indexing mode to use TSX-based SP                       |
| 3.20 |Added :SPX to return :SP-1 for #X :SPX                             |
| 3.15 |Negative AIS instructions reset the :AIS symbol                    |
| 3.12 |:AIS is set to the #SP/#SP1/#SPAUTO offset                         |
| 3.11 |Added :AIS internal symbol and #AIS directive                      |
|      |Added warning: Stack structure requires "AIS #?"                   |
| 3.10 |Added stdout for displaying to the console                         |
| 3.03 |Added warning for SP symbols outside #SP[AUTO]                     |
| 3.02 |#iftos corrected to match v3.01 change                             |
| 3.01 |Symbol :: is alias for 0-:SP when in #SP1 mode                     |
```

```
| 3.00 |Added #PSP to update PSP value with current :SP                |
|      |Fixed #SPADD to not cancel SP1 mode                             |
|      |Added #HOMEDIR directive                                        |
|      |Added NEGHX extra mnemonic                                      |
|      |Added #iftos directive [(expr + :SP) = 1]                       |
| 2.50 |Added #X for X-indexed offset                                   |
|      |Added #SPCHECK and related code                                 |
|      |#SPAUTO with parameter #OFF# turns off SPAUTO                   |
|      |#PUSH/#PULL now save/restore #SPCHECK value                     |
|      |Added :SPCHECK internal symbol returns the #SPCHECK value       |
|      |Added :PSP internal symbol returns (:SP-:SPCHECK)               |
|      |Added ,PSP (Preserved SP) indexed mode for +:psp                |
|      |Help screen beautified and made more complete                   |
|      |Added :: as alias for 1-:SP for easy definition                 |
|      |Added :SPFREE (:PSP) to return :SP-:SPCHECK                     |
|      |Added NEGXA extra mnemonic                                      |
|      |Added XGAH and XGHX extra mnemonics                             |
|      |Added #SPADD which adjusts :SP by a signed value                |
|      |Added SEXA extra mnemonic, sign-extends A into XA               |
| 2.40 |Added automatic SP adjustments for Push / Pull                  |
|      |Added directives #SPAUTO (cancel with #SP)                      |
| 2.30 |Added PSHXA and PULXA internal macros                           |
|      |Removed :1 alias for :SP1                                       |
| 2.22 |Fixed OptRtsXX warning for CALL/RTC cases                       |
| 2.20 |Change SP offset processing and fixed rare bug                  |
| 2.12 |Added ,ASP (Absolute SP) indexed mode does -:SP1                |
|      |Added ,LSP (Local SP) indexed mode does -:SP                    |
| 2.11 |Bug fix in #SP/#SP1 processing - word -> long                   |
| 2.10 |Added :1 internal symbol (alias for :SP1) ***REMOVED LATER***|
| 2.02 |Added :SP1 internal symbol to return the zero-based SP offset|
| 2.01 |Added #SP exp to offset SP indexed instructions                 |
|      |Added :SP internal symbol to return the one-based SP offset  |
| 2.00 |Added #SP1 and #SP directives                                   |
|      |Added ABS (Absolute Value of A) internal macro                  |
| 1.80 |Added :AB (AddressBytes) internal symbol                        |
|      |#PUSH and #PULL save and restore CurrentSeg                     |
|      |Fixed cross-page jumping warnings                               |
|      |Added :PAGE_START & :PAGE_END internal symbols                  |
| 1.70 |Made Date an atomic operation for consistency                   |
|      |Added #UNDEF and made EXP use 'SET', not 'EQU'                  |
|      |EXP only exports during PASS1                                   |
|      |Allowed strings up to long in EQUs                              |
|      |Division by zero in { ... } expressions shows ???               |
|      |(S)igned { ... } expressions auto-adjust by size                |
|      |Added { ... } formats up to (9) for 32-bit                      |
|      |#WARN and #NOWARN control warnings from code                    |
| 1.60 |Corrected error message "Symbol does not start"                 |
| 1.59 |Added :CYCLES for counting cycles                               |
| 1.58 |Added option -U to define output directory                      |
| 1.57 |Conditionals IF, IFZ, IFNZ are 32-bit correct                   |
| 1.56 |Conditional directives now use full 32-bit word                 |
| 1.55 |Operator ]] returns the high word, [[ low word                  |
| 1.54 |Added CRC at end of listing file                                |
| 1.53 |Fixed LONG bug                                                  |
| 1.52 |Added LONG for 32-bit constant data                             |
|      |Added 32-bit label definitions                                  |
| 1.51 |Added PUSH and PULL extras                                      |
| 1.50 |Internal changes only                                           |
| 1.49 |BugFix: SET was updating label sequence                         |
| 1.48 |Made TP option usable with console output                       |
| 1.47 |Made TP option usable with common ERR files                     |
| 1.46 |Added RESET extra instruction for forcing reset                 |
| 1.45 |Internal changes only                                           |
| 1.44 |#PULL now flushes S19 if forceS2 state changes                  |
| 1.43 |"MMU is disabled" converted to Error                            |
| 1.42 |Added #S2 (-S2+) and #S1 (-S2-) directives                      |
| 1.41 |Made S2 appear only in MMU or ForceS2 mode                      |
| 1.40 |Operator [[ returns the low word                                |
|      |Allowed 24-bit constant strings                                 |
|      |Made S19CRC use linear address when -Z enabled                  |
| 1.39 |CRC and S19CRC counters initialized for pass 2                  |
| 1.38 |Added S2/S8, CALL/RTC, #MMU/NOMMU, FAR                          |
|      |Added option -MMU[+/-]                                          |
|      |Added { ... } formats (5) and (6) for 24-bit                    |
|      |Added option -S2 for forcing S2 production                      |
|      |Added option -Z[+/-] for linear addresses S19                   |
|      |Added #IFMMU and #IFNOMMU                                       |
|      |When MMU is off, CALL -> JSR and RTC -> RTC &                   |
|      |-J option and #JUMP #CALL to control the same                   |
| 1.37 |Added S19CRC and made it display in summary                     |
| 1.36 |Added error for unresolved { ... } expression                   |
```

```
| 1.35 |Added { ... } expression recognition in EQU strings          |
| 1.34 |Added CRC display in completion status line                   |
| 1.33 |#CRC shows error for undefined symbols                        |
| 1.32 |Added #CRC directive and :CRC internal symbol                 |
| 1.31 |Added SETN pseudo-op                                          |
| 1.30 |Improved #VARIABLE violation message                          |
| 1.29 |Q option also disables #Message output                        |
| 1.28 |Added NEXP pseudo-op                                          |
| 1.27 |NEXT expression evaluates as Integer. Added show error.       |
| 1.26 |Added optional expression to NEXT                             |
| 1.25 |Added system variables YEAR, MONTH, and DATE                  |
| 1.24 |Allowed repeaters to be expressions                           |
| 1.23 |Added #VARIABLE directive                                     |
| 1.22 |Added NEXT pseudo-op                                          |
| 1.21 |Added { ... } format modifiers (1) thru (4)                   |
| 1.20 |Change default format for embedded expressions                |
| 1.19 |Added expression evaluation in strings                        |
| 1.18 |Added expression evaluation in #Message etc                   |
| 1.17 |Added SET pseudo-instruction                                  |
| 1.16 |Added PSHCC and PULCC extra instructions                      |
| 1.15 |Optimized AAX for 1 less byte, 2 less cycles                  |
| 1.14 |Fixed bug with remaining Integer() parts                      |
| 1.13 |Internal changes only                                         |
| 1.12 |Added extra mnemonic XGAX (exchange A with X)                 |
| 1.11 |Optimized LSLHX from 8/14 to 6/10 bytes/cycles                |
| 1.10 |Made #PUSH/#PULL save/restore #HCSON/#HCSOFF                  |
| 1.09 |Fixed cycle counts and DBNZ ,x mode recognition               |
|      |Fixed bugs with EOR                                           |
| 1.08 |Use shorter indexed modes when possible                       |
| 1.07 |Fixed bug with STHX ??,sp code                                |
| 1.06 |Added support for HCS08 (#HCSON/OFF #IF[N]HCS)                |
| 1.05 |Fixed cycle counts for several instructions                   |
| 1.04 |Added THX and TXH extras                                      |
|      |Added LSLHX extra (for multiplying HX by two)                 |
|      |Added ADDHX #nnnn extra                                       |
| 1.03 |Added errorlevel 5 for "no files found"                       |
| 1.02 |Added AAX extra to add A to H:X                               |
|      |Added INX and DEX extras for INCX and DECX                    |
| 1.01 |Added = as synonym to EQU                                     |
|      |Changed VECTORS segment default to $FFDE                      |
| 1.00 |Original (based on ASM5 v1.04) Started: 2001-07-06            |
+------+--------------------------------------------------------------+
```

Reference Guide

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 filespec is a relative path and begins with a colon (`:`) then it will be relative to the `_asm_` or Fossil root (in that order).
- If the file extension for a source filespec is omitted, the extension `.ASM` is assumed (see description of the `-R.ext` option below).
- Assembler errors may be redirected to errfile using standard DOS output redirection syntax. This capability may be used in conjunction with, or as an alternative to the `-E+` option.
- Any label can hold a value that is 32-bit long. Even though the CPU cannot understand numbers larger than 16-bit for data or addressing (MMU not withstanding), the ability to have 32-bit labels allows keeping constants that are larger than 16-bit for use in later constant calculations. Decimal numbers are signed; the largest number is +/-2147483647. Hex or binary numbers are unsigned and can go up to the full 32-bit value (2^{32}-1). For example, a symbol holding the crystal frequency of operation can be expressed with Hz detail to be used later to derive other constant values (such as bps rates or cycle-based delays).
- 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)
- 5 - The assembler did not find any files to assemble

Option	Default	Description
`-Bname`		Use 'name' as the base name for generated files, just once. If an optional extension is given which must not be one of the standard ASM8 extensions (.LST, .MAP, .SYM, .EXP, .ERR) then that extension will be used instead of S19 for the object file.
`-C[±]`	`-C-`	Label case sensitivity: + = case sensitive *See also* `#CASEON` and `#CASEOFF`
`-Dlabel`	`[:expr]`	Use up to one hundred times to define symbols for use with conditional assembly (IFDEF and IFNDEF directives). Symbol case depends on initial or subsequent -C option state. 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. If specifying the same symbol multiple times, only the most recent definition is considered. Also, in that case the repeated symbol overrides the previous one so there is no additional spot taken in the table. Prior to v11.40 repeated symbols would be processed as new independent definitions possibly causing related errors/warnings. 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-mnemonic. 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.
`-F:symb`		During assembly, if it finds the given symbol (or part of it when the special character `*` is present anywhere in the search string), it prints a 'Hint' message showing the file and line number where that symbol defines or redefines its value. This is very useful for debugging hard to locate errors of where exactly in the source code a symbol acquired its value. You may use either : or = with this option after the -F. You may use this option up to 100 times. This option cannot be saved with the -W switch
`-F:num`		During assembly, if it finds the given address (in decimal or hex format the way the assembler understands numbers), it prints a 'Hint' message with the file and line number where that memory address was occupied by either data or code. This is very useful to help you resolve overlap type errors. You may use either : or = with this option after the -F. You may use this option up to 100 times. This option cannot be saved with the -W switch
`-F2[±]`	`-F2-`	Forces a *P&E Micro* 16-bit map when in MMU mode. Useful to overcome bugs in certain *P&E Micro* products that do not handle MMU addresses correctly. This option cannot be saved with the -W switch
`-FD[±]`	`-FD-`	When on, the assembler uses a fake/fixed date (specifically, Jan 1, 2011) for the internal symbols :YEAR, :MONTH, and :DATE. It does not falsify the date shown on the listing header, however. This option is useful to let you always get the same S19 CRC value (shown both at the end of the listing file, and on the command-line next to each successfully assembled file), even if you use the :YEAR, :MONTH, and :DATE internal symbols in your source code, which based on the assembly 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 of the most recent S19 CRC produced with the `-FD` option, let's you know if something has *perhaps, inadvertently* changed. Without the `-FD` option, you can't be sure if it's just the date that changed, or something else. **WARNING:** Do NOT include this option in batch or makefiles that compile your programs automatically, or you risk producing consistently misdated firmware. It should only be used for manual verification purposes. It's not by accident this option cannot be saved with the -W switch.
`-FE[±]`	`-FE-`	Converts warnings to error messages. This option cannot be saved with the -W switch
`-FH[±]`	`-FH-`	Forces hidden macros in listings (`#HIDEMACROS`) and ignores all `#SHOWMACROS` directives. This option cannot be saved with the -W switch
`-FI[±]`	`-FI-`	Forces display of included filenames as hints. This is useful to let you see not only the actually included files but also the order of inclusion together with the respective file count. This option cannot be saved with the -W switch
`-FL[±]`	`-FL-`	Ignores all `#LISTOFF` or `#NOLIST` directives. This option cannot be saved with the -W switch
`-FM[±]`	`-FM-`	Ignores all `#MAPOFF` directives. This option cannot be saved with the -W switch

Option	Default	Description
-FN[±]	-FN-	Hides line numbers in listings. Useful to compare two listings for content using most DIFF tools even when there are minor differences in line numbering. This option cannot be saved with the -W switch
-FQ[±]	-FQ-	Does not show the assembly progress. Slightly better speed. For use with IDEs and makefiles. This option cannot be saved with the -W switch
-FW[±]	-FW-	Converts warnings to harmless messages. No error code is returned and warnings are not counted. This option cannot be saved with the -W switch
-FX[±]	-FX-	Enable macro line number display in FATAL, ERROR, WARNING, MESSAGE, and HINT directives. The default is off for less cluttered display. This option cannot be saved with the -W switch
-G[±]	-G-	Enable warnings under MMU mode when using CALL with local or JUMP/BRANCH with global. This option can be saved with the -W switch. *See also* #GCALL and #LCALL
-H[±]	-H+	Enables or disables the display of all user hints. When disabled, only hints that are enabled by the system (such as with the -FI option to show include files) will be generated. When enabled, all hints will be displayed.
-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	?2;?1;?a	Define default INCLUDE directory root(s). Relative path files will be tried relative to this directory (or each directory in the given directory list, searched from left to right). Multiple directory roots may be specified in the form of a list. A directory root list is separated by semi-colons when on Windows, or colons when on Linux. The * character can be used as a placeholder for the current text of this option (e.g., when you want to add extra directories either before or after the current ones without having to specify the whole thing again.) *Since v9.56* two special placeholders may be used in the directory list specification: ?1 which refers to the path of the main file, and ?2 which refers to the path of the current file. Since the main file is now searched last, and the current file's path is no longer searched by default, to get the same behavior as was the default in previous versions, you should run this to convert the current path to the new format but with compatible behavior: -i.;?1;?2;* -w (for Linux, use -i.:?1:?2:* -w). *Since v9.58*, FOSSIL source control management users can use the ?F (case-insensitive) placeholder to specify the root directory of the repository. This allows for truly portable installations of your code base. *Since v9.61*, users can use the ?A (case-insensitive) placeholder to specify the assumed root directory, which must contain a filename named _asm_ (lowercase in Linux) of zero length, even. This allows for truly portable installations of your code base. If _asm_ is present, the produced MAP files will use relative file paths (unless the -U option is used). This switch does not affect absolute path file definitions. Both the INCLUDE and the IF(N)EXISTS directives are affected by this switch.
-J[±]	-J+	Effective only while the MMU is disabled: When on, CALL/RTC instructions are treated as if they were JSR/RTS instructions, respectively. When off, CALL/RTC instructions produce errors. 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. *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.
-LLnum	-LL19	Define the maximum recognizable Label Length from the legacy 19 characters up to an absolute maximum of 50 characters. *See also the directive* #MAXLABEL.
-LS[±]	-LS-	Create a *.SYM symbol list (one for each file assembled). May be useful for debuggers that do not support the *P&E Micro* 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[±]	-M+	Create a *.MAP (one for each file assembled).* .MAP files created may be used with debuggers that support the *P&E Micro* source-level map file format.
-MMU[±]	-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 (my own public domain format) -MTP: Generate proprietary *P&E Micro*-style map file
-O[±]	-O+	Enables these four warnings: S19 overlap, RMB overlap, Violation of MEMORY directive, and Violation of VARIABLE directive.
-P[±]	-P+	When on it tells the assembler to stop after Pass 1 if there were any errors. Provides for faster overall assembly process and less confusion by irrelevant side errors of Pass 2. Warnings do not affect this.

Option	Default	Description
-Q[±]	-Q-	Specifies quiet run (no output) when redirecting to an error file. 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.
-Rn	-R74	Specifies maximum length of S-record files. The length count n includes all characters in an S-record, including the leading S and record type, but not the CR/LF line terminator. Minimum value is 12 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[±]	-REL+	Allows generation of BRA/BSR instead of JMP/JSR optimization warnings when enabled. *See also* OPTRELON/OPTRELOFF
-RTS[±]	-RTS-	Allows generation of JSR followed by RTS subroutine call optimization warnings when enabled. *See also* OPTRTSON/OPTRTSOFF
-S[±]	-S+	Generate *.S19 object file (one for each file assembled).
-S2[±]	-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[±]	-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. Since you only need a single S9 record in the final S19 file, you can use this option to not produce S9 records for all but one of the files that will be merged together to produce a single object file with a single S9 record. This option cannot be saved with the -W switch. *Example*: Application and bootloader merging. Assuming you merge the first with the second (in that order), the bootloader should be assembled as usual, and the application with the -S9 option in effect.
-SH[±]	-SH-	Include dummy S0 record (header) in object file (only if -S+).
-SP[±]	-SP-	When enabled, the operand part of an instruction is stripped of spaces before parsing. In this case, possible comments must begin with semi-colon. *See also* SPACESON/SPACESOFF
-T[±]	-T-	Makes all errors look like Borland errors (useful to fool certain IDEs).
-Tn	-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 n^{th} character.
-Ux	*(none)*	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. Using this option will also force absolute file paths in the produced MAP files.
-X[±]	-X+	Allow recognition of extra, non-68HC08-standard mnemonics and simulated index modes in source files. *See also* EXTRAON/EXTRAOFF
-Z[±]	-Z-	Convert the paged addresses (PAGE:ADDR16) to linear (extended) address in the produced S19 file(s). In effect, all addresses within the ranges $xx8000-$xxBFFFare 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. Warning: The ambiguous case of $8000-$BFFF is treated as PAGE0, which is converted to linear addresses: $000000-$003FFF. If you want to place something at PAGE2, position it (ORG) at $028000, which will convert to linear address $008000.
-WRN[±]	-WRN+	Enables or disables the display of all warnings. When enabled, only warnings that aren't disabled individually will be generated. When disabled, it overrides local warning options (such as -REL and -RTS).
-W	*(none)*	Write options specified on command line to the asm8.cfg file. The user-specified options become the default values used by ASM8 in subsequent invocations. Filespec(s) on the command line are ignored. Assembly of source files does not take place if this option is specified.
-WW	*(none)*	Same as -W but removes licensing information from the configuration file (e.g., for public distribution).

Source File Pseudo-Instructions

Pseudo-Op	Description

Pseudo-Op	Description
`[label] ALIGN expr`	*Case 1*. If no label is present, it aligns the current location counter to be a multiple of the given expression. *Case 2*. If a label is present it aligns the value of that label to be a multiple of the given expression. (In this case, however, it does nothing to the current location counter.) It issues an error if the label is not already defined. **COMPATIBILITY ISSUE WITH VERSIONS PRIOR TO 8.30:** Prior to version 8.30, the optional label would be assigned the current location counter value after the alignment. The label could not be defined earlier, or you would get an error. With v8.30 and later, you get an error if the label is not already defined by the time ALIGN is reached because the new behavior requires a previous definition so it can align the existing value of the label. This makes it easy to catch all incompatible ALIGN statements written for the previous version(s). If you get an error, simply move the label after the ALIGN statement.
`DB string¦expr[,...]`	Define Byte(s). expr may be a constant numeric, a label reference, an expression, or a string. DB encodes a single byte in the object file at the current location counter for each expr encountered (using the LSB of the result) or one byte for each character in strings.
`DS blocksize`	Define Storage. The assembler's location counter is incremented by blocksize. Forward references not allowed. No code is generated.
`DW expr[,...]`	Define Word(s). expr may be a constant numeric, a label or an expression. expr is always interpreted as a word (16-bit) quantity, and is stored in the object file at the current location counter, high byte followed by low byte.
`END [expr]`	Provided for compatibility. The END directive cannot be used to terminate assembly; ASM8 always processes the source file to the end of file. If expr is specified, the word result of the final END directive is encoded in the S9 record of the object file. If the expr 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 object file (no S9 record is produced in that case).
`[Label] ENDM`	Ends definition of a macro. The optional label is defined prior to the ENDM processing, which means it is in the same scope as the rest of the macro.

`label DEF expr[,size]`	Assigns a DEFault value to a label. In other words, this is a conditional EQU. It only assigns the label if the label is currently undefined.
	
	`LABEL DEF EXPR`
	
	is equivalent to:
	
	`#IFNDEF LABEL`
	`LABEL EQU EXPR`
	`#ENDIF`
	
	Note 1: The value that appears in the listing file is the actual (effective) value of the label, which may be different from the value of the expression, since the assignment may not occur.
	
	Note 2: If the expression is missing, a relevant error message will appear prompting you to use the `-D` option to define one. This is useful to specify (somewhere near the top of your main/include file) those symbols that require predefinition at a higher level by simply typing a list of `label DEF`, one for each required definition. This will produce more relevant errors than those you normally get when referring to an undefined symbol in some expression inside the file as the user is notified that those symbols are specifically expected to be predefined, and not just lacking definition by mistake.
	
	Note 3: Option `-?` will allow interactive definition of `label DEF` variant.

`label EQU expr[,size]`	Assigns the value of expr to label. *See also* `EXP` and `SET`
`label EXP expr[,size]`	Assigns the value of expr to label. This is similar to `EQU` but with the following difference: Labels defined thus will be included in the .EXP file as regular SETs. 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.

Pseudo-Op	Description
`FAR expr[,...]`	Define 24-bit word(s) when the MMU is enabled. expr may be a constant numeric, a label or an expression. expr is always interpreted as a 24-bit quantity, and is stored in the object file at the current location counter in big-endian order. If, however, the MMU option is disabled, `FAR` is treated as `DW`.
`FCB string¦expr[,...]`	Form Constant Byte(s). Same as DB.
`FCC string¦expr[,...]`	Form Constant Character(s). Same as DB.
`FCS string¦expr[,...]`	Form Constant String. Similar to FCC, but automatically appends a terminating null (0) byte to the end of the string defined (for ASCIZ strings). If an empty string is given, only the ASCIZ terminator is inserted.
`FDB expr[,...]`	Form Double Byte(s). Same as DW.
`LONG expr[,...]`	Form 32-bit long word(s). expr may be a constant numeric, a label or an expression. expr is always interpreted as a 32-bit quantity, and is stored in the object file at the current location counter in big-endian order.
`MacroName MACRO comments`	`MACRO` begins the definition of a new macro.
`... REMACRO ...`	`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.
`MERROR [text]`	Combines an `#ERROR` directive followed immediately by an `MEXIT`, which is commonly found in macros. This can only be used inside macros.
`MEXIT [expr]`	Causes an unconditional early exit from a macro expansion. (Normally, used inside a conditional block.) If the optional expression (without any forward references) is present, its value will be placed in the `:MEXIT` internal variable. If the expression is missing, the current value of `:MEXIT` will not be changed, allowing for cascaded return values from nested macros.
`MSUSPEND`	`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.
`MRESUME`	Suspending a macro preserves the current macro state (parms, counters, etc.) just like nested macros do to protect the parent macro's state, but it allows for code outside any macros to be assembled in place of the `MSUSPEND` keyword, as if it were part of the macro (except that it is actually assembled outside the macro, so none of the macro-only features can be used, and none of the macro limitations apply; for example, normal use of `#INCLUDE` is possible, as well as definition of new macros, etc.) This makes it much easier to create nestable macros that emulate block structures, than by using two separate macros (one for block begin, and one for block end) and trying to keep them synchronized. *Note*: When a nested macro is suspended, all macros leading to the currently executing macro are indirectly suspended as a side effect. `MRESUME` can be used only from outside any macros to resume execution of the most recently suspended macro.
`MSTOP [#ALL#]`	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.)
`MSTOP [text]`	When used outside a macro, it causes the most recent suspended macro to stop being suspended. When the optional `#ALL#` parameter is used, then all nested suspended macros are stopped (become no longer suspended).

`MSTR index[,index]*`	`MSTR` tests each one of the specified indexed macro parameter text for being a string, and, if not a string, it changes it to one using the appropriate delimiters based on the contents of the parameter text.
	It is equivalent to the following sequence (but repeated for each specified index n):
	`#IFPARM ~n.~`
	`#IFNOSTR   ~n.~`
	`MSET    n,\@~n.~\@`
	`#ENDIF`
	`#ENDIF`

Pseudo-Op	Description
`MSET index[,text]`	`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.
`MSET #['charset']`	A second variation of `MSET` (added in v8.90) allows to unite all parameters into one, and optionally split back into as many parameters using a user-defined character set given as string. For example, `MSET #` will unite all current macro parameters into just one (using the currently active macro parameter separator.) `MSET #'abc'` will first unite all parms into one and then split into separate parms for every occurrence of characters a, b, or c, while skipping over parenthesized parts, and quoted strings. If the string after `#` has just one character, this will be eliminated (as it will be treated as the new parm separator.) If it has more than one character (even the same character twice), the parm will be split right before the character, and the found character will be the first character of the next parm. This feature allows you to re-arrange the parameters based on a different separator.
`MDEF index[,text]`	`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, near the top of the macro).
`MSWAP index,index`	`MSWAP` simply swaps the text of any two parameters. (Swapping a parameter with itself has no effect.) As an example for `MSWAP`, in macros with multiple single operands, you can use it to bring the working operand always in, say, `~1~`, which may be simpler to use than the equivalent `~{:loop}.~` from inside a loop.
`MDEL index`	`MDEL` deletes the current macro's index-ed parameter. Note: This is different that using `MSET` without the text parameter to delete the content of a parameter placeholder. `MDEL` deletes the actual parameter location, which means all following parameters will move down one location. For example, `MDEL 1`, will move `~2~` to `~1~`, `~3~` to `~2~`, and so on, for all parameters that follow.
`MTRIM index[,index]*`	`MTRIM` will trim all non-string spaces from the respective parameter(s). *Note*: In all cases, index is any expression that doesn't contain forward references.
`MREQ ind[,ind]*[:errmsg]`	Checks each of the specified macro parameters (separated with commas) for null value (empty). If the parameter is null, an appropriate internal error message is displayed, and the macro expansion is terminated at that point. If the optional errmsg parameter is present (which must follow a colon), this error message will be displayed instead of the default error message. This can be used to specify which macro parameters are required, and print an error message, if these parameters are null. If more than one of the specified parameters are null, the message will repeat for each one of them. You may use `MREQ` multiple times, perhaps, once for each parameter, so that you can have a unique error displayed for each parameter. *Note*: `ind` is any expression that doesn't contain forward references. `errmsg` is any text. If `ind` contains an internal variable (such as `:LOOP`), it must be enclosed in `{ ... }` because the colon is also used as the beginning of the errmsg.

Pseudo-Op	Description

MTOP [limit expr]	Causes an immediate unconditional jump to the top line of the current macro, while incrementing the :LOOP counter. It can be used either alone or within conditionals. The advantage to using MTOP over @~0~ (a macro call to self) is that whatever parameters were passed in the macro do not need to be specified again as the macro is never exited. Also, no counters are incremented, except for :LOOP. This means, however, that $$$ based labels (which are unique to a macro invocation) are still in the same scope as before the MTOP command since no new macro has been invoked.
	
	If the optional limiting expression (containing only non-forward references) is present, its value will be compared to the :LOOP counter and MTOP will execute only if the current value of :LOOP is less than the value of the expression. Example (shift word right one or more times):
	
	`lsr.w     macro     Address[,Count]`
	`mdef      2,1    ;default Count=1`
	`lsr       ~1~`
	`ror       ~1,~+1~,1~`
	`mtop      ~2~`
	`endm`
	
	As another example, an expression like the one that follows can be used to loop while the next parameter is not null:
	
	`mtop :loop+:{:loop+1}`

MDO [start expr]	MDO and MLOOP work together to form a local DO ... LOOP inside a macro. Note: MDO and MLOOP cannot be nested because MLOOP always matches the most recent MDO of the current macro. MDO simply marks the current line (i.e., the line containing the MDO keyword) as the beginning of a local loop, and (re)initializes the :MLOOP counter to one (1), or to the value of the non-forward expression, if one is present.

MLOOP [limit expr]	MLOOP causes an immediate unconditional jump to the line following the most recent MDO keyword, while incrementing the :MLOOP counter (not to be confused with the :MACROLOOP or :LOOP counter). If no MDO was used up to this point in the macro, MLOOP jumps to the top of the current macro (just like MTOP would), but it only affects the :MLOOP counter (whereas MTOP only affects the :LOOP counter).
	
	If the optional limiting expression (containing only non-forward references) is present, its value will be compared to the :MLOOP counter and MLOOP will execute only if the current value of :MLOOP is less than the value of the expression. Example (multi-byte addition):
	
	`add.m macro Op1,Op2,Ans[,Size]`
	`mdef 4,1 ;default size = 1`
	`#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`
	
	As another example, an expression like the one that follows can be used to loop while the next parameter is not null:
	
	`mloop :mloop+:{:mloop+1}`

Pseudo-Op	Description
`label NEXP symbol[,expr]`	Assigns the current value of symbol to label as if with `EXP`. Then, it increments the value of symbol by one (as if with `SET`) or, if the optional expression is present, by the value of that expression. Useful for defining a series of symbols based on a common starting value. *Note*: symbol is a single label and not an expression. *See also* `NEXT`,`SETN`
`[label] NEXT symbol[,expr]`	Assigns the current value of symbol to label as if with `EQU`. Then, it increments the value of symbol by one (as if with `SET`) or, if the optional expression is present, by the value of that expression. Useful for defining a series of symbols based on a common starting value. *Note*: symbol is a single label and not an expression. Since v9.41, a special case of `NEXT` is when the label to the left is missing. In that case,`NEXT`is used as an anonymous placeholder that simply increments the symbol to the right, as usual. *See also* `NEXP`,`SETN`
`ORG [[s19_expr],]expr`	Sets the assembler's location counter for the active segment. Code generated after this directive will be assembled starting at the location specified by expr. If s19_expr is present, then the S19 file runs with an offset from the actual location counter. This allows for different segments of code to be assembled at the same physical address but, obviously, be placed in different addresses in the loadable S19 file. The current offset is available in the `:OFFSET` internal symbol. To cancel any offsets without changing the current position, simply give ORG followed by a single comma without any expressions.
`label PROC`	`PROC` first advances the `@@` local label counter, and then it assigns the value of the program counter (`*`) to label. This allows using symbols locally for a specific section of code (e.g., a subroutine). The symbol to the left of `PROC` is always in the new scope. The name of the symbol is stored in the `procname` macro placeholder. Each time `PROC` (or `#PROC`) is encountered, the assembler increments an internal 32-bit local symbol counter. Symbols containing `@@` anywhere inside their name (except at the very beginning) at least once (for example, `Loop@@`) will have the `@@` part replaced with a special control character (different from what is used with macro local `$$$`) and the current value of the internal local symbol counter (similar to `$$$` with macro local labels). Up until a `PROC` or `#PROC` is encountered in the program, the `@@` is not treated specially (i.e., the `@@` is not converted to a special number). This makes this feature compatible with code written prior to its introduction. The current value of the corresponding internal counter can be found in the internal symbol `:PROC` while the maximum proc number can be found in the internal symbol `:MAXPROC`
`[label] ENDP`	`ENDP` is optional and marks the end of the corresponding PROC. Its use allows one to nest procs (e.g., for code coherency as when keeping a subroutine close to the actual point of use). The optional label is defined prior to the `ENDP` processing, which means it is in the same scope as the rest of the proc. *See also* `#PROC` and `#ENDP`
`RMB blocksize`	Reserve Memory Byte(s). Same as `DS`.

Pseudo-Op	Description
`label SET expr[,size]`	Assigns the value of expr to label even if label is already defined with a different value. This is similar to `EQU` but allows making multiple re-definitions. The value set will be used until another `SET` pseudo-instruction or to the end of the assembly process. Warning: Careless, or simply wrong use of this directive can lead to multiple side errors or warnings (please note this is a two-pass assembler). Using a forward `SET` defined symbol may lead to problems, as the value used will be the one from the last `SET` definition, which is not necessarily the one we want. Correct behavior is guaranteed if any symbols re-defined with `SET` are used only after each new re-definition, otherwise, the first reference in Pass 2 will use the value from the last re-definition in Pass 1. *Example of wrong use:* 1.`lda #Value ;we expect 123, actual is 234` 2.`Value equ 123` … 3.`lda #Value ;we expect 234, actual is 123` 4.`Value set 234` Value in line 1 will be 234 (the last known value from Pass 1) while Value in line 3 will be 123 (most recent value in current Pass 2). *Example of correct use:* 1. `Value equ 123` 2. `lda #Value ;we expect 123, actual is 123` … 3. `Value set 234` 4. `lda #Value ;we expect 234, actual is 234` *See also* `EXP` and `EQU`
`label SETN symbol[,expr]`	Assigns the current value of symbol to label as if with `SET`. Then, it increments the value of symbol by one (as if with `SET`) or, if the optional expression is present, by the value of that expression. Useful for (re-)defining a series of symbols based on a common starting value. Note: symbol is a single label and not an expression. *See also* `NEXP`,`NEXT`

Pseudo-Op	Description
[label] PIN [expr[,bitnum]]	Assigns a specially combined value of expression and bit number to a single label. This special purpose SET variant is meant for defining port pins, single register bits, or single flag bits found in zero page (i.e., zero MSB of address) port/register/variable definitions that are commonly used with BSET, BCLR, BRSET, and BRCLR instructions. When defining a symbol thus you can then use it with simplified B[R]CLR and/or B[R]SET instructions, i.e., without specifying the address and bit as two separate parts of the instruction operand. For example: LED pin PORTA,3 can be used with BSET LED instead of BSET 3,LED or BSET 3,PORTA. Also, BRCLR LED,LedIsOff instead of BRCLR 3,LED,LedIsOff. This makes it very simple to re-assign pins to a different location by simply changing the pin definition and no other code. Another important feature of the pin pseudo-op is that it can be used to take the next bit value automatically. The first use within a program requires at least an expr to be present. If a bit number is not given, zero (the least significan bit) is assumed. Each subsequent use without parameters will assume the next bit position for the same address, up to bit 7, after which it will issue errors. Until now, special macros were used for PIN, B[R]CLR, and B[R]SET to emulate the above functionality by using three separate symbols, one for the port/register/variable address, one for the bit number, and one for the bit mask, and having to use macros for B[R]CLR/B[R]SET instead of the real instructions. (The macros are still useful when dealing with non-zero-page entities, however.) Comparing pins has now become straight forward as only a single symbol needs to be compared. For example, #if LED_ALARM = LED_POWER checks if two symbols are defined on the exact same address/bit. Making an alias of a pin is achieved with a simple equ. For example, assuming LED_RED pin PORTA,0 and later LED_FAULT equ LED_RED will make both LED_RED and LED_FAULT symbols refer to the exact same address/pin combination. With the current internal pin encoding, to get just the register/variable address you need to get the LSB of the symbol (e.g., [LED or [[LED). However, this is done automatically for instructions, so you can use LDA LED instead of LDA [LED. To get the bit/pin number, use]]LED. And, to get the bit mask, use @LED which is future safe in case the internal encoding ever changes (unlikely). For example, to flip the LED bit (normally using XOR), you would do this sequence: LDA LED, EOR #@LED, STA LED. Now, you can change LED to a different port and/or pin, and your code will continue to function correctly. It's also simple to refer to other related ports. For example, assuming the DDR symbol holds the offset between PORTA and DDRA, one can change LED to output by doing BSET LED+DDR assuming of course the expression LED+DDR still falls within zero-page memory. A pin all by itself will act as a placeholder for a future definition for that bit position. It simply skips that bit number. If label is missing but expr is present, pin only defines the starting address/pin for subsequent pin pseudo-ops but does not use up a bit position. This is useful when you have a series of pin definitions that must happen only under certain #if conditions. If even the first flag (pin) definition (bit 0) is under condition, it would be very cumbersone to define the flags as you would need to know what other flags may have been defined already (to avoid leaving empty bit positions). Now, you simply let the assembler deal with all that complexity. Example: pin PORTA, #ifdef NEED_SOUND, SOUND pin, #endif followed by similar constructs for other pins/bits/flags. All pins have size 1 as they naturally occupy part of a whole byte. When #Export-ing a pin definition it will appear as a regular set pseudo-op but it will be interpreted correctly. Actually, you could use equ or set to define a pin 'manually' but using the special pin pseudo-op provides additional benefits and protection from common errors.

Assembler 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 option 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.

Directive	Description

Directive	Description
#AIS [symbol]	#AIS checks the current value of the :SP internal variable against the most recent AIS instruction's value, and issues a warning if the two numbers do NOT differ by the exact value in the symbol (note: a plain symbol, not an expression), indicating a possible stack frame definition error (assuming correct placement of the relevant directives). The warning also shows the correct AIS instruction that is required to correct the problem. This directive makes it very easy to correct the numeric value in AIS instructions to match the following stack frame definition (normally made using the internal :: symbol in the various #SPAUTO modes, and the NEXT/SETN method for defining records/structures.) This is useful to prevent having to define the stack frame before the 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. v8.70+: If, however, a symbol is not specified, then this directive simply resets the value of the :AIS internal symbol to the current :SP value difference (as if a negative AIS instruction had been used in its place). This can be used when no actual AIS instruction is used (for example, a series of PSHx instructions are used, but we want to use the :AIS variable later on to de-allocate any local variables.) The associated :AIS symbol returns the difference between the current :SP and the value saved during the most recent AIS 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. #PUSH and #PULL will save/restore the value of this setting. Example use:

```
                    #spauto

Subroutine          ais       #-4                 ;local data

?                   set       ::
?Parm1              setn      ?,2
?Parm2              setn      ?,2

                    #ais      ?                   ;check frame definition
```

Directive	Description (continued)
#CALL	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.
#CASEOFF	When #CASEOFF is in effect, all symbol references that follow are converted to uppercase internally before they are searched for or placed in the symbol table. Equivalent to the -C- command line option.
#CASEON	When #CASEON is in effect, symbol references are NOT internally converted to uppercase before they are searched for or placed in the symbol table. Equivalent to the -C+ command line option.

Directive	Description (continued)
#CRC expr	The two CRCs (user and S19) maintained by the assembler are 16-bit each, and they are updated only during PASS2 by each produced user code/data byte that is put into the S19 file. The starting CRC value for both CRCs is zero. With this directive you can alter the user CRC value at any time (either before the very first byte of code/data to produce a different CRC for the same firmware, or several times in between to skip certain volatile sections, for example). The computed CRCs are available by accessing the internal symbols `:CRC` and `:S19CRC` The formula used for the 16-bit CRC calculation is very simple to be easily implemented even in tiny bootloaders: `16BitCRC := 16BitCRC + 16BitAddress*8BitData` `:S19CRC` is mostly useful with the `END` directive (`END :S19CRC`) as it is not affected by the `#CRC` directive. An S19 loader can check the overall integrity of the S19 file. `:CRC`, on the other hand, is mostly useful for checking code after it has been loaded into the MCU, at each reset, for example. Please note that for both CRCs all `$00` bytes do not affect the calculation while, for the user CRC only (`:CRC`), all `$FF` bytes are intentionally skipped. This allows for the CRC in an S19 file (which does not necessarily fill a contiguous block of memory) to match the CRC computed by the MCU over a complete block of memory without the MCU bootloader knowing in advance the actual addresses used within that block, provided any unused bytes are in the erased state. As a side effect, however, any `$00`->`$FF` or `$FF`->`$00` alterations in the file cannot be detected with the user CRC.
#CYCLES [expr]	First, the optional expression is calculated using the current values of any internal symbols. Then, the current value of `:CYCLES` is copied to `:OCYCLES`. Finally, the internal `:CYCLES` counter is set to zero (if the optional expression is missing), or to any arbitrary value (the result of the expression). This directive can also be used inside macros to restore the cycle counter of surrounding code, if the macro cycles should be counted in a special way, or not at all.
#DATA [expr]	Activation of the DATA segment. Default starting value is `$F000`.
#DROP macro[,macro]*	Undefines one or more macros. If a macro is not currently defined, a warning will be issued (to protect from possible typing errors). To drop all macros (global and local) with a single command, use `*` (asterisk) in place of the macro name. There is no warning if no macros found. To drop all local macros (for the current file only) with a single command, use `?*` (question mark followed by asterisk) in place of the macro name. There is no warning if no local macros found. If used from inside a macro, and that macro is dropped, the macro will terminate at that point. The rest of the macro will not be processed. The special macro named `?` (just a single question-mark) is to be used ad-hoc, and it is automatically dropped (without warning) at each new redefinition. You may also drop it with `#DROP` but only need to do so if you want to force errors in later use of the macro, so you can easily locate them. You cannot drop macros that are currently active above the current macro level (e.g. nested macros leading to current one.) The `#!DROP` variant will suppress warnings for undefined macros.
#EEPROM [expr]	Activation of the EEPROM segment. Default starting value is `$0000`.
#EJECT	*See* `#PAGE`

Directive	Description (continued)
#ELSE [IFxxx]	When used in conjunction with conditional assembly directives (#IF, #IF[N]DEF, #IF[N]Z, #IFMAIN, #IFINCLUDED, etc.), code following the #ELSE directive is assembled if the conditional it is paired with evaluates to a not-true result. Optionally, you can follow with another IF directive (of any kind) to create a 'chained' condition check, like: `#IF ... #ELSE IF ... #ELSE IF ... #ELSE ... #ENDIF` The optional IF should not start with a # or $ directive symbol but it should be separated with at least one space.
#ENDIF	Marks the end of a conditional-assembly block. Conditional assembly statements may be nested if they are properly blocked with #ENDIF directives.
#ERROR [text]	When encountered in the source, the assembler issues an error message in the same form as internally-generated errors, using the text specified, prefixed with USER:
#EXIT [expr]	If no expression is present, it immediately exits the current #INCLUDE file. (Does nothing if used inside a main file.) If the optional expression is present (normally though, this might be just a single label), the exit occurs only if the expression is defined (as if when checked with #IFDEF). This can be used in the top of #INCLUDE files, like so: #EXIT *COMMON* In this example, the first time this file is included, the symbol *COMMON* is undefined, so the #EXIT is ignored. Consequent times this file is included, it exits upon hitting the #EXIT directive. *Note*: Due to how #INCLUDE files are counted internally, and there being a limit on how many total files you can #INCLUDE, it's better when working with larger projects that you do not #INCLUDE a file at all when already processed, rather than #INCLUDE it and #EXIT it. *For this reason, in most cases, prefer* #USES
#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 symbol	Export one or more symbols (as if with *EXP*). File-local symbols cannot be exported. If a symbol is not currently defined, a warning will be issued. The #!EXPORT variant will suppress warnings for undefined symbols. Since v11.80 you can follow the symbol with an alias symbol separated by either space or equals sign. For example, #EXPORT Start Begin will export symbol Start with the name Begin instead.
#EXPORT symbol|expr,symbol	Export one or more symbols (as if with *EXP*). File-local symbols cannot be exported. If a symbol is not currently defined, a warning will be issued. The #!EXPORT variant will suppress warnings for undefined symbols. Since v11.90 if you follow the symbol with an alias symbol separated by either space or equals sign the first part can be an expression. For example, #EXPORT ROM_END-ROM ROM_SIZE will export symbol ROM_SIZE with the value of the expression ROM_END-ROM.
#FATAL [text]	Similar to the #ERROR directive, but generates an assembler fatal error message and terminates the current file assembly (processing will continue with possible further files in the command line supplied file list).
#GCALL	Effective only while the MMU is enabled: When active, CALL instructions to local labels or JUMP/BRANCH instructions to global labels produce respective warnings, unless prefixed with !. *See also the directives* #LCALL, #MMU, #NOMMU Equivalent to the -G+ command line option.
#HCSOFF	Disables the HCS08 instruction set mode. *See also* #IFHCS #IFNHCS #HCSON Equivalent to the -HCS- command line option.

<table>
<tr><th>Directive</th><th>Description (continued)</th></tr>
<tr><td>#HCSON</td><td>Enables the HCS08 instruction set mode.
See also #IFHCS #IFNHCS #HCSOFF

Equivalent to the -HCS+ command line option.</td></tr>
<tr><td>#HOMEDIR [path]</td><td>Makes the specified path the current home directory. Although this cannot affect where any output files will go, it does make a difference on where any following relative #INCLUDE files will be searched. Relative file path specifications will now be relative to the directory specified by the #HOMEDIR directive, including any relative #INCLUDE references in nested include files.

If [path] is missing, the original main file path is restored.</td></tr>
<tr><td>#IF expr1 cond expr2</td><td>Evaluates expr1 and expr2 (which may be any valid ASM8 expression) and compares them using the specified cond conditional operator. If the condition is true, the code following the #IF operator is assembled, up to its matching #ELSE or #ENDIF directive.

Cond may be any one of: < <= = >= > <>

The condition is always evaluated using unsigned arithmetic.

If a symbol referenced in expr1 or expr2 is not defined, the statement will always evaluate as false.
The #!IF variant will suppress warnings for undefined symbols.</td></tr>
<tr><td>#IFDEF [!]expr</td><td>Attempts to evaluate expr, and if successful, assembles the code that follows, up to the matching #ELSE or #ENDIF directive. This directive is used to test if a specified symbol has been defined or, if prefixed with !, undefined. Symbol(s) referenced in expr must be defined before the directive for the result to evaluate true (e.g., forward references will evaluate as false). #IFDEF without an expr following will always evaluate to False.

alt-0166 (¦) or double pipe (||) ORs separated conditions while double ampersand (&&) ANDs separated conditions. AND has higher precedence than OR.</td></tr>
<tr><td>#IFEXISTS fpath</td><td>Checks for the existence of the file specified by fpath (using the same rules as those used for #INCLUDE directives) and assembles the code that follows if the specified fpath exists.</td></tr>
<tr><td>#IFHCS</td><td>Assembles the following code if the assembler is in the extended HCS08 instruction set mode. *See also* #IFNHCS #HCSON #HCSOFF</td></tr>
<tr><td>#IFINCLUDED</td><td>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</td></tr>
<tr><td>#IFMAIN</td><td>Assembles the code that follows if the file containing this directive is the main (primary) file being assembled. *See also* #IFINCLUDED.</td></tr>
<tr><td>#IFMDEF macro</td><td>#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.</td></tr>
<tr><td>#IFNOMDEF macro</td><td>#IFNOMDEF does the opposite check.</td></tr>
<tr><td>#IFMMU</td><td>Assembles the code that follows if the MMU option is enabled. *See also the directives* #MMU,#NOMMU, and #IFNOMMU</td></tr>
<tr><td>#IFNOMMU</td><td>Assembles the code that follows if the MMU option is disabled. *See also the directives* #MMU, #NOMMU, and #IFMMU</td></tr>
<tr><td>#IFPARM text [= text]
#IFPARM text [== text]
#IFNOPARM text [= text]
#IFNOPARM text [== text]

Aliases:

#IFB same as #IFNOPARM
#IFNB same as #IFPARM</td><td>Normally used inside macros. If text is non-blank, assembles the code that follows, up to the matching #ELSE or #ENDIF directive. This directive is used to test if a specified macro parameter has been defined. #IFPARM without text following (after macro expansion) will always evaluate to False. text is usually a parameter placeholder (e.g., ~1~).

You can also make a case-insensitive (using the = sign) or case-sensitive (using the == sign) comparison of the parameter to a specific text string (with or without quotes, depending on your intent) by separating the two text strings with an ‘equals’ (=), or double-equals (==) sign, depending on the desired case-sensitivity. For example,

#IFPARM ~1~ = * tests if parameter one is a plain asterisk (normally used to indicate the current location pointer.)

#IFNOPARM performs the opposite test.</td></tr>
<tr><td>#IFSPAUTO</td><td>Assembles the code that follows, up to the matching #ELSE or #ENDIF directive, if the assembler is currently in #SPAUTO (automatic SP adjustment) mode. *See also* #SPAUTO #SP</td></tr>
</table>

Directive	Description (continued)
#IFSTR text	Normally used inside a macro. If text is a quoted string, assembles the code that follows, up to the matching #ELSE or #ENDIF directive. This directive is used to test if a specified macro parameter is a string. #IFSTR without text following (after macro expansion) will always evaluate to False. text is usually a parameter placeholder (e.g., ~1~).
#IFNUM text	Normally used inside a macro. If text represents a number, assembles the code that follows, up to the matching #ELSE or #ENDIF directive. This directive is used to test if a specified macro parameter is a number. #IFNUM without text following (after macro expansion) will always evaluate to False. text is usually a parameter placeholder (e.g., ~1~).
#IFNONUM text	#IFNONUM performs the opposite test.
#INCLUDE fpath	Includes the specified fpath 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 255 levels (the main source file counts as one level). By default, fpath specifications are referenced relative to the location of the current source file, and if not found there, relative to the assembly's main source file, and finally, relative to the optional user added 'root' project file (having the filename _asm_ regardless of content). One only has to use forward looking paths (i.e., no ../ components) irrespective of the 'depth' of the current source directory. Included files will be found either under the current directory or in a forward relative location from the assumed root point. This approach has proven over the years to work exceptionally well for practically all projects, and rarely, if ever, the user has to provide a modified -I option path list.
#USES fpath	#USES is an alternative, slightly different method to include a file. It will #INCLUDE the file specified (using the same file-finding rules as #INCLUDE) but only if the same file path has not been included (via #INCLUDE or #USES) at least once, already. #USES is useful for creating #INCLUDE file dependencies (normally, from a higher level to a lower level; e.g., an analog temperature sensor driver module #USES the A/D driver module, but not the other way around). This allows directly #USING (*an alias for #USES*) only the module of interest in your application, and it should take care to use whatever other modules it requires (in a recursive sort of way). If another included module in the same application #USES the same lower-level module, it will not be included a second time. This is similar to the common technique used to prevent multiple inclusions of the same file, but only have it included the first time it is referenced. Normally, the #IFNDEF ... #ENDIF block is found inside the file, meaning the assembler must enter the file before it 'knows' it doesn't need it. The advantages with #USES, however, are: (1) you do not need a specific symbol definition for each file, and (2) you never enter an already included file (which would use up a sometimes precious file count towards the maximum number of #INCLUDE files.) #IFNDEF MODULE MODULE *... your module code goes here* #ENDIF Bi-directional, or circular co-dependencies (e.g., file A depends on file B, while file B depends on A) are possible in some cases, and then they require some extra attention in the respective files' internal organization, or it could not work as you might have expected, and leave you confused by 'spurious' errors. In general though, you should try to avoid them. Also, you cannot use #USES in place of #INCLUDE for modules that must be intentionally included multiple times (e.g., including the same SCI driver module, once for each hardware SCI available), although you could use #USES to include a file that itself does #INCLUDE the same file multiple times. *Note*: The assembler will only generate a standard error (not an assembly-terminating fatal error) if a file specified in a #INCLUDE (or #USES) directive is not found. The #IFEXISTS and #IFNEXISTS directives may be used in conjunction with #FATAL if termination of assembly is desired under such conditions.
#IFNDEF [!]expr	Evaluates expr and assembles the code that follows if the expression could NOT be evaluated, usually as the result of a reference to an undefined symbol or, if prefixed with !, defined symbol. This directive is the functional opposite of the #IFDEF directive. alt-0166 (¦) or double pipe (||) ORs separated conditions while double ampersand (&&) ANDs separated conditions. AND has higher precedence than OR.
#IFNEXISTS fpath	The opposite of #IFEXISTS; code following this directive is assembled if the specified fpath does NOT exist. The -Ix 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 expr	Evaluates expr and assembles the code that follows if the expression evaluates to a non-zero value. #IFNZ always evaluates to false if expr references undefined or forward-defined symbols.

Directive	Description (continued)
#IFTOS expr	If top-of-stack evaluates expr+:SP (+:SP is implied) and assembles the code that follows if the expression is equal to one (when in #SP[AUTO] modes), or zero (when in #SP1 mode), i.e., expression points to top-of-stack in all modes. #IFTOS always evaluates to false if expr references undefined or forward-defined symbols. Useful mostly in #SP[AUTO] modes.
#IFZ expr	Evaluates expr and assembles the code that follows if the expression is equal to zero. #IFZ always evaluates to false if expr references undefined or forward-defined symbols.
#JUMP	Effective only while the MMU is disabled: When active, CALL/RTC instructions are treated as if they were JSR/RTS instructions, respectively. 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. *See also the directives* #CALL, #MMU, #NOMMU Equivalent to the -J+ command line option.
#LCALL	Effective only while the MMU is enabled: When active, CALL instructions to local labels or JUMP/BRANCH instructions to global labels do not produce any warnings. *See also the directives* #GCALL, #MMU, #NOMMU Equivalent to the -G- command line option.
#LISTOFF #NOLIST	Turns off generation of source and object data in the* .LST file for all lines which follow this directive. Useful for excluding the contents of #INCLUDE files in the *.LST file.
#LISTON #LIST	Enables generation of source and object data in the .LST file for the source code following this directive. Has no effect if list file generation is disabled (-L- command line option in effect).
#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 [@@]	#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 [@@]	#MCF2 is almost the same as #MCF but it doesn't have the restriction where macros named the same as instructions require the @macro format from within macros. This is the most 'dangerous' of all available modes, since it is always the macro which has precedence. If you need to be certain you use a real instruction and not a possible macro with the same name, you MUST use the !instruction format.

Directive	Description (continued)
#@MACRO [@@]	#@MACRO turns off this option. This is the default setting when a new assembly begins. In this mode, you can only invoke macros with the @ prefix. This is the recommended mode for most normal applications. Hint: The macro is normally invoked as an instruction, which means its name must appear after column one. Regardless of the current macro mode, when a macro call is made using the default @macro (or %macro) format, its invocation can start even in column one, since it can't ever be a symbol that starts with one of these two characters [@ and %]. *Note*: If the optional @@ parameter is provided to any of the four directives mentioned above, macro chaining is effectively disabled, and any otherwise 'chained' calls now become truly nested calls (as if the @@macro format is used at all times a macro is called). *WARNING*: Macros written based on the default 'chain' behavior may no longer operate the same (since non-@@ macro calls include an implied following mexit). To simulate the same behavior, when the @@ option is active, make sure you add an MEXIT command after each otherwise 'chained' macro call. By the way, this will make the macro work the same way regardless of the @@ sub-mode being in effect or not. When the @@ sub-mode is in effect, you still need to observe the various calling methods based on which of the three macro modes you're in. To cancel the @@ sub-mode, simply give any of these directives without it.
#MAXLABEL number	Define the maximum recognizable Label Length from the legacy 19 characters up to an absolute maximum of 50 characters. *See also the command-line option -LL.* #PUSH and #PULL will save/restore the value of this setting.
#MEXPORT macro[,macro]*	Export one or more macros in the EXP file (if one is produced). File-local macros cannot be exported. If a macro is not currently defined, a warning will be issued. The #!MEXPORT variant will suppress warnings for undefined symbols.
#MLIMIT [expr]	Sets the maximum macro nesting limit to the value of the optional expression. If no expression follows the default value of 100 is used. This value should be more than adequate for nearly all cases. Minimum value is zero (which practically disables macro call nesting). Maximum is 10000 (ten thousand). *Note*: Macro nesting uses extra memory during assembly. You should avoid using macro nesting if the same functionality can be achieved by using macro chaining, or even the most efficient simple looping (MTOP pseudo-instruction).
#MLISTOFF #NOMLIST	Turns off generation of source and object data in the *.LST file for all macro body lines which follow this directive. Useful for excluding the body of macros in the .LST file.
#MLISTON #MLIST	Enables generation of source and object data in the *.LST file for all macro body lines following this directive. Has no effect if list file generation is disabled (-L- command line option in effect). This is the default setting.
#HIDEMACROS	Note: These two directives work only when the* -LC- (List Conditionals = OFF) command-line option is in effect. #HideMacros treats all macro-specific keywords (the @macro call, MEXIT, MTOP, ENDM) the same as 'conditional' directives only for the purposes of display in the listing. So, when -LC- is in effect, they won't appear in the .LST file at all. This leaves only the expanded macro contents. When this directive is in effect, it is no longer possible to know where a macro begins or ends, or how many times it iterates itself.
#SHOWMACROS	#ShowMacros (re-)enables normal display. The default setting when a new assemby begins is #SHOWMACROS. *Note*: The corresponding macro definitions will not display at all, regardless of the -LC mode. #PUSH and #PULL will save/restore the value of this setting.
#MAPOFF	Suppresses generation of source-line information in the *.MAP file for the code following this directive. Symbols which are defined following this directive are still included in the *.MAP file.
#MAPON	Enables generation of source-line information in the *.MAP file for the code following this directive. #MAPON is the default state when assembly is started when map file generation is enabled (-M+ command line option).

Directive	Description (continued)
#MEMORY addr1 [addr2] #MEMORY #OFF#	Maps a memory location (or range, if addr2 is also supplied) of object code and/or data areas as valid. Use multiple directives to specify additional ranges. Any code or data that falls outside the given range(s) will produce a warning (if the -O option is enabled) for each violating byte. Very useful for segmented memory devices, etc. Addr1 and addr2 may be specified in any order. The range defined will always be between the smaller and the higher values. The special keyword #OFF# removes all current definitions. *See also* #VARIABLE
#MESSAGE [text]	Displays text on screen during the first pass of assembly when this directive is encountered in the source. Messages or hints are not written to the error file. They are meant to inform the user of options used or conditional paths taken.
#HINT [text]	#HINT cannot be masked with the -Q+ option, but it can be masked with the -H- option.
#HINTS	Allow user generated hints to display. Use the -H- to hide them. System hints will always display.
#NOHINTS	Disallow user generated hints from display. Use the -H+ to show them. System hints will always display.
#MMU	Enable the MMU features (e.g., CALL/RTC instructions, 24-bit addresses and expressions). *See also the directives* #NOMMU, #IFMMU, and #IFNOMMU Equivalent to the -MMU+ command line option.
#NOMMU	Disable the MMU features (e.g., CALL/RTC instructions, 24-bit addresses and expressions). *See also the directives* #MMU, #IFMMU, and #IFNOMMU Equivalent to the -MMU- command line option.
#NOWARN	Turns warnings off. Equivalent to the -WRN- command line option. *See also* #WARN
#OPTRELOFF	Disable BRA/BSR instead of JMP/JSR optimization warnings. Equivalent to the -REL- command line option.
#OPTRELON	Enable warning generation when an absolute branch or subroutine call (JMP or JSR) is encountered that could be successfully implemented using the relative form of the same instruction (BRA or BSR). This option is on by default. Equivalent to the -REL+ command line option.
#OPTRTSOFF	Disable RTS-after-JSR/BSR optimization warning (default). Equivalent to the -RTS- command line option.
#OPTRTSON	Enable warning generation when a subroutine call (JSR or BSR) is immediately followed by a RTS. This option is off by default. Command-line option -RTS+ does the same thing.
#PARMS [ch|SPC]	Allows changing the delimiter used to separate macro parameters when invoking the macro. If char is defined the new delimiter will be the same as char. If there is no character following the directive, the default parameter delimiter (a comma) will be used. To use a regular space as a parameter separator, the [char] part of the command should be the special keyword SPACE (case-insensitive). #PUSH and #PULL will save/restore the value of this setting.
#PPC	#PPC (stands for Preserve PC) simply keeps a copy of the current :PC value to be used later by the :PPC internal symbol. #PUSH and #PULL will save/restore the value of this setting.
#PROC	Advances the @@ local label counter. Nullifies the contents of the ~procname~ macro placeholder. *See also* PROC and #ENDP
#ENDP	Closes the corresponding PROC section. *See also* PROC and ENDP
#PSP	#PSP (stands for Preserve SP) simply keeps a copy of the current :SP value to be used later by the :PSP internal symbol. The :PSP symbol returns the difference between the then current :SP and the value saved with this directive. This can be used to de-allocate just the number of stack bytes that were pushed in between. This is only meant for use in #SPAUTO mode, which automatically adjusts the current value of the :SP internal symbol. #PUSH and #PULL will save/restore the value of this setting.

Directive	Description (continued)
#RENAME oldname,newname	Renames a macro from its current (old) name to a new name. An error message is issued if the old name is not a defined macro, the new name is a defined macro, or either name is an invalid symbol name.
#REMACRO oldname,newname	`#REMACRO` is similar to `#RENAME` but it does NOT check if the new name exists. If it exists, there will now be one extra instance of that macro name. *Note*: Only the most recently defined macro of the same name is visible when more than one macro definitions share the same name. `#DROP`-ping the macro always drops the visible instance, making a possible previous instance now visible. *Tip*: An example of where `#RENAME` might be useful: Say, you have a library (or OS system) macro that is called many times in your application, but you want to modify that macro's behavior just for this one application. Your options are: [1] Write a new (differently named) macro, and change all calls from the old macro to new macro. Problem: If some of these calls are inside shared library code, you can't change those calls, as it will affect other applications using those macros, as well. Too much work, and error prone. [2] Alter the library macro to include the new behavior. Problem: Other applications may not like the new behavior. [3] Use `#RENAME` in your application to have the old library macro change name just for this application's sake. Then, use the original name to write a brand new compatible macro but with the new behavior. It is also now possible for the new macro to 'borrow' the functionality of the old macro (by calling it internally as needed), so the new macro doesn't necessarily have to repeat the whole original macro body. This allows for an easy way to extend or replace any general-purpose library macros for each application, separately. Example for `#REMACRO` that allows front-ending a previous macro to add code before and after the original macro call.

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

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

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

                    @a
                    #Message  ----------------------------
                    @a
```

Directive	Description (continued)
#S19FLUSH	Forces the immediate termination of an S-record line when encountered, rather than waiting for the record to reach the size specified by the `-Rn` command line directive. This directive may be used to make identification of the end of code blocks easier when viewing the *.S19 file.
#S19RESET	Resets the S19 processor. Any used address ranges will be forgotten allowing the same ranges to possibly be used again. This directive may be used to combine multiple normally overlapping S19 files into one.
#S19WRITE [text]	Flushes the current S19 record as with the `#S19FLUSH` directive, and then writes the optional text message into the S19 file on a line by itself. This directive may be used to add arbitrary text inside the S19 file, such as comments, special processing loader directives, etc.
#S1	`#S1` reverts to normal, which produces S1 or S2 records based on address.

Directive	Description (continued)
#S2	#S2 forces 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. Equivalent to the -S2- and -S2+ command line options.
#SIZE symbol[,expr]	Assigns the value of the expression to the size attribute of the specified previously-defined label. If no expression is present, then the difference between the current location and the label is used (as if the expression was: *-LABEL) which is the most common use of this directive. You can access the size attribute at a later time by using the internal symbol ::symbol (where symbol is the symbol whose size you want to get.)
#SP [expr]	#SP *without any expression cancels* #SP1 *and* #SPAUTO *modes (reverts to default/normal operation).* *#SP* followed by any expression (including a zero value) sets the :SP offset to the value of that expression but does not affect the current #SPAUTO mode. The assembler always starts in plain #SP mode (no offsets).
#SP1 [expr]	#SP1 *automatically adds one to all SP-indexed offsets. It does this without affecting the current value of the :SP internal symbol. 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)!
#SPAUTO [expr][,expr]	#SPAUTO (or its shorter alias, #SPA) will automatically adjust the offset based on the instructions used. All push and pull instructions (including the extra ones) as well as all AIS instructions will automatically adjust the offset by as many bytes as required by each instruction. Use the #SP directive (without any 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 without changing the current SP offset.) Since v8.90 #SPAUTO takes an optional second argument (any valid constant expression). If this value is specified, then the assembler (while in SPAUTO modes) will produce warnings when the stack depth increases beyond the value of the given expression. This value will remain active until it is explicitly turned off by using -1 in a subsequent #SPAUTO directive (e.g., #SPAUTO ,-1). The current value of this expression you can find in the internal variable :SPLIMIT The maximum actual stack depth used will be in the :SPMAX variable which is reset only when :SPLIMIT is rewritten with a new value. This allows to check the maximum stack depth for a single routine, a collection of routines (e.g., in a module), or the whole application.
#SPADD [expr]	#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. 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. Manual alterations of the stack size, however (such as when you push an extra byte per loop iteration) cannot be automatically detected as the assembler will not follow your code's logic. In those cases, you'll have to adjust the offset 'manually' using #SPADD and an appropriate offset, like so: #SPADD LOOPCOUNT-1 #PUSH and #PULL will save/restore the current setting of all modes of this option. *See also internal symbols* :SP and :SP1 *and the simulated indexed modes* ,ASP and ,LSP

Directive	Description (continued)
`#SPCHECK`	`#SPCHECK` checks the current value of the `:SP` internal symbol against the last used `#SPAUTO` value (found in `:SPCHECK` internal symbol), and issues a warning if the two numbers do NOT match, indicating a possible unbalanced stack situation (assuming correct placement of the relevant directives). The current difference between `:SP` and `:SPCHECK` is found in `:SPFREE` (e.g., use with `AIS #:SPFREE`) The warning also shows the number of bytes by which the stack is off. This can be used as a first-line of defense against unbalanced stack coding errors, especially in situations where there is heavy manipulation of the stack, and a visual inspection may prove confusing. Positive numbers indicate the stack contains so many extra bytes. Negative numbers indicate the stack is missing so many bytes. Hint: If you do not wish to use the `#SPAUTO` function for a particular section of code (or anywhere in your program) you can still temporarily place the `#SPAUTO` directive at the beginning of a code section to check, and the `#SPCHECK` at the end of the same code section, until you verify there are no related compilation warnings. Then you can remove the two directives (possibly even with the use of conditional directives), and continue with other coding work. *See also* `#SPAUTO`
`#X [expr]`	When `#X` is enabled (i.e., followed by a non-zero signed offset), all X-indexed instructions will have that offset value automatically added to them (on top of whatever offset is actually specified with the instruction). This has a lot of potential uses, such as pointer adjustments (after `TSX`), or anytime the same constant needs to be added to a series of X-indexed instructions within a block of code. `#PUSH` and `#PULL` will save/restore the current setting of this option. The assembler always starts in plain `#X` mode (no offsets). *See also internal symbol* `:X` *and the simulated indexed mode* `,AX`
`#UNDEF symbol[,symbol]*`	Undefines one or more symbols. If a symbol is not currently defined, a warning will be issued (to protect from possible typing errors). Careless, or simply wrong use of this directive can lead to multiple side errors or warnings (please note this is a two-pass assembler). If you simply want to redefine the value of a symbol, prefer using the `SET` pseudo-instruction, rather than using `#UNDEF` followed by a repeated symbol definition. `#UNDEF` can be used, for example, to completely remove unrelated or conflicting conditionals. The `#!UNDEF` variant will suppress warnings for undefined symbols.
`#PAGE`	Outputs a Form Feed (ASCII 12) character followed by a Carriage Return (ASCII 13) in the .LST file just before displaying the line that contains this directive.
`#PUSH`	Pushes on an internal stack the current segment and the current settings of the following directives: `MAPx`, `LISTx`, `CASEx`, `EXTRAx`, `SPACESx`, `OPTRELx`, `OPTRTSx`, `[NO]WARN`, `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`
`#PULL`	Pulls from an internal stack the most recently pushed options. *See also* `#PUSH`
`#RAM [expr]`	Activation of the RAM segment. Default starting value is `$0080`.
`#ROM [expr]`	Activation of the ROM segment. Default starting value is `$C000`. This is the default segment if none is specified.
`#SEGn [expr]`	Activation of the SEGn segment (n is a number from 0 through 9). Default starting value for all ten segments is `$0000`.
`#TABSIZE n`	Specifies the field width of tab stops used in the source file. Proper use of this directive ensures that the *.LST files generated by ASM8 are formatted in the same way as your source files appear in your text editor. This directive overrides the setting of the `-Tn` command line option for the source file(s) in which it is encountered.

Directive	Description (continued)
#TEMP [expr]	#TEMP simply assigns any value (possibly the result of an non-forward expression) to the internal general-purpose :TEMP variable. If no expression follows #TEMP, :TEMP is zeroed. :TEMP can be used any time in lieu of defining any 'helper' symbol for intermediate calculations (either inside or outside macros). The only restriction is that :TEMP always refers to the most recent #TEMP directive, so it cannot be used to look forward. Although :TEMP is a single variable, its use is transparent in relation to macros. In other words, changing :TEMP from within any macro does not affect the value of :TEMP outside that macro, regardless of nesting level. Although macros inherit their initial value of :TEMP from their higher level (either a caller macro, or normal code), they do not affect their parent's :TEMP value, so you can use it without worrying about side effects from any intermediate macro calls. :TEMP is also assigned indirectly when used as label with any of the following directives/pseudo-ops: NEXT, NEXP, SETN, and #AIS
#TEMP1 [expr]	#TEMP1 works exactly like #TEMP providing additional storage.
#TEMP2 [expr]	#TEMP2 works exactly like #TEMP providing additional storage.
#TRACEON	#TRACEON enables generation of source-line information in the .MAP file for any code found in the body of macros following this directive. The map info is generated in such a way that while tracing the debugger will display the actual source of the macro. This can be used globally (to affect all macro invocations), inside a specific macro (to debug that one macro), or around a specific macro invocation (to debug that one macro call.)
#TRACEOFF	#TRACEOFF turns this option off making macros appear as a single line in the debugger. #TRACEOFF is the default state when assembly is started.
#VARIABLE addr1 [addr2]	Maps a location (or range, if addr2 is also supplied) of variable allocation area (normally in RAM) as valid. Use multiple directives to specify additional ranges. Any RMB or DS definitions that fall (fully or partially) outside the given range(s) will produce a warning (if the -O option is enabled) for each such definition. Addr1 and addr2 may be specified in any order. The range defined will always be between the smaller and the higher values.
#VARIABLE #OFF#	The special keyword #OFF# removes all current definitions. *See also* #MEMORY
#VECTORS [expr]	Activation of the VECTORS segment. Default starting value is $FFC0.
#WARN	Turns warnings on. Equivalent to the -WRN+ command line option. *See also* #NOWARN
#WARNING [text]	Similar to the #ERROR directive, but generates an assembler warning message instead of an error message.
#XRAM [expr]	Activation of the XRAM segment. Default starting value is $0100.
#XROM [expr]	Activation of the XROM segment. Default starting value is $8000.

Macros

- 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.
- A possibly undefined macro can be called by prefixing the macro name with ! Example: @!macro will call macro only if it is defined. It is equivalent to the sequence:
  ```
  #ifmdef macro
  @macro
  #endif
  ```

- 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 file-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.) You can also have the macro decide how parameters are separated (*see `MSET`*).
- 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.) You can have more than nine parameters but to access them you'll have to use the `~n.s.l~` form *with default `s` and `l` parts*, e.g., `~10.~` accesses the 10th parameter.
- `~Cn~` where n is a number from 0 thru 9 is equivalent to `~{n}.{:loop}.1~`
- `~-Cn~` where n is a number from 0 thru 9 is equivalent to `~{n}.{:{n}-:loop+}.1~`
- `~cn~` where n is a number from 0 thru 9 is equivalent to `~{n}.{:mloop}.1~`
- `~-cn~` where n is a number from 0 thru 9 is equivalent to `~{n}.{:{n}-:mloop+}.1~`
- 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 use). With conditional assembly (e.g., `#IFPARM ~#~`) one can treat the `~1~` parameter differently, assuming

immediate mode.

- Similarly, the placeholder `~#n~` (where n is a number from 1 thru 9, zero also accepted but it is pointless) returns the parameter part after a possible `#` sign, if one is present. This allows getting an immediate mode type parameter in a form (stripped of the `#` symbol) that can be used in expressions (for example, in an `#IF` directive expression). Note: If no `#` is inside the parameter, `~#n~` is the same as `~n~` alone.

- 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 *by chaining to itself*, must use the exact 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 chained (or nested) macro invocation (a macro invocation occurring from inside a macro) then the text value of `~label~` is not changed from the original macro's. This way, a macro can chain to itself (for looping), or another macro and still have the `~label~` placeholder expand correctly. Note: The length of the actual text inside `~label~` can be found in the internal variable `:label`.

- 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.

- Similarly, 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 `#RENAME` a macro from within it and then need to restore the actual name the macro had when entered, using `#REMACRO`). This allows front-ending any macro with additional functionality.

- The special placeholder `~text~` (case-insensitive) returns the current temporary text parameter of the current macro. This is a 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 as 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 the macro definition is located (`~mfilename~` etc.), which is not necessarily in the current file.

- The placeholder `~@~` is an alias for the full list of placeholders separated by • (starting from 1). Useful if you want to pass all parameters to another macro without explicitly rewriting them. The sequence produced by `~@~` is: `~1~•~2~•~3~•~4~•~5~•~6~•~7~•~8~•~9~`

- The placeholder `~@@~` is an alias for the full list of placeholders separated by • but starting from `~2~`. Useful if you want to pass the remaining parameters to the same macro when looping (assuming each loop only processes the first parameter, until that becomes null). The sequence produced by `~@@~` is: `~2~•~3~•~4~•~5~•~6~•~7~•~8~•~9~`

- An alternative to the above technique, or if you need to pass more than nine parameters is to first use `MSET #` from within the calling macro to unite all parameter under `~1~`, then call the other macro with just `~1~` as parameter, and finally have the receiving macro split the received parameters to as many places as needed by again using `MSET #c` where `c` is the assumed parameter separator.

- Trailing parameter separators (commas by default) and trailing commas due to macro expansion of null parameters are automatically removed. This is particularly useful when writing macros, which replace or enhance single-operand instructions. One can write the macro so that it does not require a parameter separator override during invocation, just so it can recognize a possible indexed operand. Example: `LDA ~1~,~2~` will work even if `~2~` is null, because the now 'dangling' comma after `~1~` will be automatically removed, preventing an otherwise expected syntax error. Similarly, `LDA ~1,~+1~,1~,~2~` will work for the following location pointed to by the expression in parm 1, regardless of the presence of an index in parm 1, parm 2, or at all.

- An alternative to the above method can be achieved (in most cases) by using the `~[n.p]~` format of the parameter placeholder (where `n` is the parameter number, or the case-insensitive keyword `text`) to extract the pth byte of the argument (e.g., if `~1~` contains `my_var,sp` then `~[1.1]~` will return `my_var,sp` while `~[1.2]~` will return `my_var+1,sp` but if `~1~` contains an immediate value such as `#$1234` then ~

[1.-1]~ will return #$12 while ~[1.-2]~ will return #$34 etc. For a 32-bit example, for parm #$12345678 ~[1.1]~ will return #$12 while ~[1.-1]~ will return #$56. The p number can be any integer (positive or negative) but for immediate mode parameters the sign matters, and it only works up to 32-bit if positive (i.e., 1..4), or up to 16-bit if negative (i.e., -1..-2). For immediate mode only, a number above 4 or below -2 will return #0 since that is the effective value. This ~[n.p]~ placeholder makes it particularly easy to deal in a unified way with any parameter, be it immediate, direct, extended, or indexed mode. Care has to be taken to use a negative p if working with 16-bit immediate mode values, however.

- You can use the ~n[set]i~ format of the parameter placeholder to extract the ith part of the nth parameter using the character set [set]. The character set can be given as a string of characters with or without quotes (square brackets, instead). Therefore, quotes can also be part of the character set. Example, ~2[,.]3~ will return the 3rd part of the 2nd parameter, where parts are separated by any instance of comma and dot. (Note: 'n' and 'i' are optional. If 'n' is missing, the value one is assumed. If 'i' is missing, the value one is assumed. Care must be taken not to allow both to be missing, if the character set contains a dot because it will then be interpreted as a ~[n.p]~ placeholder, which is processed earlier.) If the character set contains only a single character, then embedded strings will be skipped over, otherwise characters even inside strings will be matched by the characters in the set.

- You can use the ~n.s.l~ format of the parameter placeholder (where n is the parameter number, or the case-insensitive keyword text or label, or a constant string enclosed in quotes, s is the starting position, or a constant string to search for, and l is the needed length, or a constant string to search for but past the s position) to extract only a portion of the text of the corresponding parameter or constant. The first dot is required (to disambiguate from ~n~ type parms) even if nothing follows. The s and l are optional. If s is not entered its value is assumed to be one, so that ~1.~ is the same as ~1~ alone. If l is not entered, its value is the length from s to the end of the parameter (i.e., the remaining string). Note: The assembler forces s and l to always be within the limits of the text length. So, specifying a position past the end of the parameter text will always return the last character. To check for past-of-text, check against the :nnn length internal symbol for the specific parameter (e.g., :1 for parm one). If you need to make n, s, or l the result of an expression you can use {expr} (for example: ~1.{:loop}.2~).

  SPECIAL CASE: When inside a string, the expression will be evaluated when the string is processed by the assembler, which is after macro expansion of the various placeholders. This means we have lost our chance to expand this placeholder. But, we can use the \@ instead of quotes for strings inside a macro which contain ~n.s.l~ embedded expressions, and not only those (example: fcc \@~{PARM}.{FROM}.{LENGTH}~\@ to have it expand correctly. Because of the \@ the string does not appear as a string yet, and the expressions can be calculated during macro expansion. This way all expressions become simple constants, and the placeholder can be processed. Finally, the \@ dummy string delimiters are turned into single, double, or back quotes, depending on which one 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 had been available with the problematic 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.p]~, ~n.s.l~, ~n[set]i~, \@string\@, and ~Cn~ variations.
- During macro invocation, any parameter text may contain embedded expressions of the form {<expr>}, like one can do with strings, where 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 *(i.e., before any PC incrementing code)* 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 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 @@ mode (see the relevant section for details) . Macro-chaining will be altogether disabled, however.
- To break out of an accidental endless macro loop, press [ESC] on the command-line.
- Macro labels may be case-sensitive (depending on #CaseOn/Off directives) when defined, but are always case-insensitive when invoked

(like normal mnemonic 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.

- Virtually unlimited number of macro definitions (memory permitting.)
- Virtually unlimited size of each macro (memory permitting.)
- Unlimited number of macro invocations (all internal macro counters are 32-bit).

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
```

String Expressions and Formatting

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 word or long with leading $, - `(B)` for hex byte without leading $, - `(W)` for hex word without leading $, - `(L)` for hex long without leading $, - `(S)` for signed decimal, - `(1)` thru `(4)` (or, thru `(9)` for the 32-bit versions) for the corresponding number of decimal places after division by 10^n where n is a number from 1 to 4 (or 9), - `(X)` for expanded *(i.e., both decimal and hex)*, - `(Fn)` for space left filled, where n is optional (default is 2) and can range from 1 to 0 (0 meaning 10). - `(Zn)` for zero left filled, where n is optional (default is 2) and can range from 1 to 0 (0 meaning 10).

If no format modifier is used, `(D)` is assumed.

Some examples using this feature:

```
ROM                 equ       $F000
                    #Message  ROM is at {ROM}
```

displays: `ROM is at 61440`

Adding a format modifier has the following effect:

```
                    #Message  ROM is at {ROM(x)}
```

displays: `ROM is at 61440 [$F000]`

```
                    #Message  ROM is at {ROM(d)}
```

displays: `ROM is at 61440`

```
                    #Message  ROM is at {ROM(h)}
```

displays: `ROM is at $F000`

```
                    #Message  ROM is at {ROM(s)}
```

displays: `ROM is at -4096`

```
  #Message Clock: {:year}-{:month(z)}-{:date(z)} {:hour(z)}:{:min(z)}:{:sec(z)}
```

displays something like: `Clock: 2013-11-05 13:00:00`

It can also be used in strings, like so:

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

is equivalent to:

```
MsgVersion          fcs       'Firmware v1.01',LF
```

but it automatically adjusts 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}'`.

Internal symbols

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

- `::` (without a symbol following) returns the current *dynamically assigned* stack offset. Very useful mostly in `#SPAUTO` mode so that you can assign labels to stack contents as they are created. (Same as `1-:SP` in `#SP[AUTO]` modes, or `0-:SP` if in `#SP1` [sub-]mode.) *Note: Beginning with v5.70 if any push instruction is followed by a label, that label will be `SET` to the current `::` value (in `#ExtraOn` mode).*

- `::symbol` - a double colon followed by any previously defined symbol returns the current 'size' for the given symbol. A symbol's size is determined either automatically (e.g., `RMB` pseudo-instruction), or manually via the `#SIZE` directive.

- `:SP` returns the current offset of the `#SP` or `#SP1` directives. This value is the basis for several other internal symbols.

- `:SPLIMIT` returns the currently effective value of the `#SPAUTO` stack depth check option (i.e., the optional 2nd parameter of the `#SPAUTO` directive.) `#PUSH`/`#PULL` save/restore this value.

- `:SPMAX` returns the maximum used stack depth since the last time `:SPLIMIT` was explicitly set *even if to the same value it already had*. You can use this internal variable to find the maximum stack depth for a single routine, a collection of routines (e.g., a whole module), or even your whole application's. Keep in mind, however, that it only counts stack depth in a linear fashion, i.e., without considering possible subroutine calls, recursion, or other indirect methods of altering the stack, such as the `LDHX #STACKTOP` / `TXS` sequence. It stills offers significant insight into your code's approximate stack requirements.

- `: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 as it is more readable*, 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` and `MOV X+` instructions. (Note: there are many ways to alter the contents of the HX index register; the assembler does not 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` and `MOV X+` 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 will not 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 (or `#AIS` directive when given without any parameters). 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., after a subroutine call that passes some parameters on the stack) to free just the number of stack bytes for that specific section of code. Simply, place a #PSP directive right before stacking the subroutine parameters, call the function, and follow with AIS #:PSP to remove the parms.
- :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 Address Bytes (useful for stack offsets). It returns 2 for normal mode, or 3 for MMU mode. This can be used primarily to correctly and automatically refer to caller stack regardless of the MMU mode being active or not. It can also be used to allocate storage for pointers and to calculate offsets into a pointer table regardless of mode.
- :X returns the current offset of the #X directive.
- :YEAR returns the year at assembly time (e.g., 2021) Hint: Use :YEAR\100 for two-digit year.
- :MONTH returns the month at assembly time (e.g., 2)
- :DATE returns the date at assembly time (e.g., 26)
- :HOUR returns the hour at assembly time (e.g., 13)
- :MIN returns the minute at assembly time (e.g., 0)
- :SEC returns the second at assembly time (e.g., 0)
- :CPU returns a number representing the CPU type (6808 for older 68HC08, or 908 for newer 9S08)
- :VERSION returns the assembler version as integer with two implied decimal digits.
- :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 resets it to zero.
- :CCYCLES returns the current value of the cycles counter (v9.90+).
- :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 Macro use not allowed for this application
  #ENDIF
  ```
  placed at the end of your code).
- :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 a 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 an explicit %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` keyword pair. 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 an explicit `%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 need to use the same value more than once in the same macro invocation, you must first assign the value to some label (or `:TEMP`), and then use the label, instead. This counter is also reset with a `%macro` syntax call.
- `:0` to `:999` 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 assembly time day-of-week number, 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 larger applications).
- `:PROC` returns the current value of the internal local symbol counter. *See `PROC` and `#PROC`.*
- `:MAXPROC` returns the currently maximum value of the internal local symbol counter. *See `PROC` and `#PROC`.*
- `:LABEL` returns the length of the `~label~` placeholder's content used inside macros.
- `:TEMP`, `:TEMP1`, and `:TEMP2` return the current value of each of these internal user-defined assembly-time variables. *See the directives `#TEMP`, `#TEMP1`, and `#TEMP2` for more details.*
- `:MAXTEMP`, `:MAXTEMP1`, and `:MAXTEMP2` return the current maximum value of their respective `:TEMP` variable. They are reset to zero with `#TEMP`, `#TEMP1`, or `#TEMP2` directives, respectively. They are preserved with respect to macros the same as with `:TEMP` variables. When entering a macro, these variables are initialized to the current value of their respective `:TEMP` variable, and restore their previous value on exiting the macro.
- `:TEXT` returns the length of the current text of the `~text~` macro parameter *only from within macros*.
- `:LINENO` returns the current file line number.
- `:MAXLABEL` returns the current value of the maximum recognized label length. *See `-LL` command-line option and `#MaxLabel` directive.*
- `:MLINENO` returns the current macro line number *only from within macros*.
- `:N` returns the current macro number of contiguous arguments *only from within macros*.
- `:NN` returns the current macro number of all arguments *only from within macros*. It equals the maximum parameter index even if previous parameters are null.
- `:ANRTS` returns the address of the most recent `RTS` instruction (i.e., always points back). It produces an error if an `RTS` hasn't appeared already.
- `:ANRTC` returns the address of the most recent `RTC` instruction (i.e., always points back). It produces an error if an `RTC` (MMU mode) or either `RTC` or `RTS` (normal mode) hasn't appeared already.
- `:ROM`, `:RAM`, etc. All segment directives have a corresponding internal variable that returns the current value of that segment.
- `:OFFSET` returns the current S19 addressing offset from the physical address (see `ORG`).
- `:WIDTH` returns the current width of the console screen. Useful for formatting user messages.

- `:WARNINGS` returns the current number of warnings.
- `:ERRORS` returns the current number of errors.

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      SOME_OFFSET
                    ...
?_a_                equ       1
                    pshx
                    cmpa      1-:sp1,sp           ;absolute offset
                    cmpa      ?_a_-:sp1,sp        ;absolute offset
                    cmpa      ?_a_,asp            ;absolute offset (preferred)
                    pulx
```

All three `CMPA` 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 `SOME_OFFSET`. It is equivalent to `cmpa 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` or `#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` or `#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                 proc
                    tsx
                    lda       MyOffset,sp         ;gets A from parent stack
                    lda       MyOffset+:sp,x      ;(equivalent)
                    lda       MyOffset,spx        ;(equivalent, preferred)
```

Examine the following assembler listing 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       #-7                 ;(negative AIS marks current :SP)
          ...
          ais       #2                  ;de-allocate some locals (no marking)
          ...
          pula                          ;de-allocate some locals (no marking)
          ...
          ais       #:ais               ;deallocate stack (4 in this case)
                                        ;still left since negative AIS
          pull                          ;restore all registers
          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 previous example becomes:

```
                #spauto                         ;auto mode and zero offset

                #psp
                ldhx      #SkipTo               ;new instruction added later
                pshhx                           ;new instruction added later
                ldhx      #SkipFrom             ;new instruction added later
                pshhx                           ;new instruction added later
                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       #:psp                 ;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                           ;new instruction added later
                ldhx      #SkipFrom             ;new instruction added later
                pshhx                           ;new instruction added later
                ldhx      Size+4,sp
                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 figure it out automatically), so certain optimizations that depend on specific stack offsets may no longer work if stack ordering is later changed. For example, *divide HX by ten*:

```
                pshhx                           ;place number to divide on stack
                clrh
                ldx       #10                   ;divisor
                pula                            ;A = MSB (assuming number at TOS)
                div                             ;divide it
                psha                            ;put it back on stack
                lda       2,sp                  ;A = LSB (assuming number at TOS)
                div                             ;divide it
                sta       2,sp                  ;save it back to stack
                pulhx                           ;HX = result
```

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

                clrh
                ldx       #10                   ;divisor
                pula                            ;A = MSB (assuming number at TOS)
                div                             ;divide it
                psha                            ;put it back on stack
                lda       2,sp                  ;A = LSB (assuming number at TOS)
                div                             ;divide it
                sta       2,sp                  ;save it back to stack

                pula                            ;<<< added later to protect A
                ...
                #sp                             ;cancel auto mode and offsets
```

it will no longer work because `PULA` / `PSHA` always work on the top-most stack element (`#SPAUTO` does not alter mnemonics), which in this case is

no longer the number we want to divide (as `PSHA` before `CLRH` 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

Routine             proc
                    pshhx     number@@              ;stack & name number to divide

                    psha                            ;added later to protect A

                    clrh
                    ldx       #10                   ;divisor
        #iftos number@@
                    pula                            ;A = MSB (assuming number at TOS)
                    div
                    psha
        #else
                    lda       number@@,sp           ;A = MSB
                    div
                    sta       number@@,sp
        #endif
                    lda       number@@+1,sp         ;A = LSB
                    div
                    sta       number@@+1,sp

                    pula                            ;added later ...

                    ...
                    ais       #:ais                 ;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 stack and assign a name to the number. Finally, using conditional assembly (`#IFTOS`), 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.

Another method to stack and name the number is:

```
                    #spauto                         ;auto mode and zero offset

Routine             proc
                    pshhx
number@@            set       ::,2                  ;stack & name number to divide
```

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*).

The above example 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 *and 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 subroutine 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 *logically* 'after' the #SPAUTO directive using the :: internal symbol as a base offset. An `#SP` directive after the end of the subroutine turns all automatic SP adjustments off.

You may even re-define a stack-offset symbol (either using a 'named push' or the `name SET ::` method) 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              proc
                    pshhx     parm@@              ;stack parm and name it
                    ...
                    bsr       Child
                    ...
                    sthx      parm@@,sp           ;affects Parent's stack variable
                    rtc

;*******************************************************************************

                    #spauto   2                   ;auto mode plus RTS

Child               proc
parm@@              equ       1                   ;caller TOS before calling Child
                    ldhx      parm@@,sp           ;get parm from Parent's stack
                    pshhx     parm@@              ;stack local copy of parm and name it
                    ...                           ;(name "parm@@" now refers to Child's copy)
                    sthx      parm@@,sp           ;affects Child's stack variable
                    pulhx
                    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 and in the `code` subdirectory of the assembler ZIP archive.

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. A second 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 cycle counted 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 for a given bus speed (`BUS_KHZ` symbol should be defined accordingly):

```
                              #Cycles             ;reset cycles counter
Delay10ms           proc
                    pshhx
                    ldhx      #DELAY@@
                              #Cycles             ;grab cycles (and reset)
```

```
Loop@@              aix       #-1
                    cphx      #0
                    bne       Loop@@
                              #temp :cycles       ;grab cycles (and reset)
                    pulhx
                    rtc                           ;(normal and MMU compatible)

DELAY@@             equ       10*BUS_KHZ-:cycles-:ocycles/:temp
```

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                         ;HX = CRC

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

GetAsmCRC           proc
                    #temp     1
start_address@@     next      :temp,2
end_address@@       next      :temp,2
my_crc@@            next      :temp,2

                    push

Loop@@              ldhx      start_address@@,sp
                    cphx      end_address@@,sp
                    bhi       Done@@

                    lda       ,x
          #ifdef COP
                    sta       COP                 ;in case of many iterations
          #endif
                    beq       Cont@@
                    cbeqa     #$FF,Cont@@

                    mul                           ;low address with data byte
                    add       my_crc@@+1,sp
                    sta       my_crc@@+1,sp
                    txa
                    tsx
                    adc       my_crc@@,spx
                    sta       my_crc@@,spx

                    ldhx      start_address@@,sp
                    lda       ,x
                    thx
                    mul                           ;high address with data byte
                    tsx
                    add       my_crc@@,spx
                    sta       my_crc@@,spx

Cont@@              tsx
                    inc       start_address@@+1,spx
                    bne       Loop@@
                    inc       start_address@@,spx
                    bra       Loop@@

Done@@              pull
                    rtc

                    #sp                           ;(usually, only at end of file)
```

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
```

or, in more recent ASM8 versions, you could use the `#temp` directive (preferred):

```
                    #temp     :crc

          ;CODE/DATA TO SKIP FROM CRC CALCULATION HERE

                    #CRC      :temp
```

Expression Operators and Other Special Characters

- Expressions are evaluated in the order they are written: left to right. **All operators have equal precedence.** Many other assemblers behave the same in this respect. Although at first this may seem like a nuisance, in practice *for assembly language coding in particular* it turns out that *in most cases* this allows for less complex, easier to read expressions, and it also makes it easier to use with macro parameters. For example, instead of `lda {~1~}*2,x` one simply writes `lda ~1~*2,x` as `~1~` is already known to evaluate correctly regardless of the operators used in a possible expression inside `~1~`, such as `OffsetB-OffsetA`. Care is needed when porting code from other assemblers that don't use left- to-right operator precedence to transform all expressions, accordingly. This should be straight forward in most cases. You can use `{ ... }` to change the default precedence, such as `{1+2}*{3+4}`. The only limitation is no forward references are allowed.

- Avoid inserting spaces between values and operators (unless using `-SP+` command-line switch or the `#SpacesOn` directive together with `;` beginning comments). Otherwise, the expression will evaluate until the first space encounted and you will get unexpected results.

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`

Operator	Description
`$`	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 <)
`&&`	`#if[n]def` logical AND between terms (has precedence over logical OR).
`¦¦`	`#if[n]def` logical OR between terms.

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
`[!]...`	A placeholder `NOP` instruction. If the `!...` format is used and symbol `...` is defined, then this instruction will produce a `#HINT` followed by whatever text follows, or if no text follows some default message. If the symbol `...` is undefined, this instruction will be treated as a regular `NOP` but without causing `#MEMORY` violation warnings when `#MEMORY` is active. The `...` format will always produce a `#HINT` regardless of symbol `...` being defined or not. By placing `...` (optionally followed by an appropriate message) where the code needs your attention for further work, you can easily locate those places simply by assembling; with the `-D...` option (if `!...` is used).
`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`
`ASLH`	Same as: `PSHH` / `ASL 1,ASP` / `PULH`
`ASRH`	Same as: `PSHH` / `ASR 1,ASP` / `PULH`
`CLRHX`	Same as: `CLRH` / `CLRX`
`CMPA operand`	Same as: `CMP operand`
`CMPX operand`	Same as: `CPX operand`
`DECH`	Same as: `PSHH` / `DEC 1,SP` / `PULH`
`DEX`	Same as: `DECX`
`INCH`	Same as: `PSHH` / `INC 1,SP` / `PULH`
`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`)

Mnemonic/Syntax	Description
`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`)
`LSLH`	Same as: `PSHH` / `LSL 1,ASP` / `PULH`
`LSLHX`	Logical shift left of HX *useful for table offset adjustment* Same as: `PSHH` / `LSLX` / `ROL 1,SP` / `PULH`
`LSRH`	Same as: `PSHH` / `LSR 1,ASP` / `PULH`
`NEGHX`	Same as: `PSHH` / `COM 1,SP` / `PULH` / `COMX` / `AIX #1`
`NEGXA`	Same as: `COMX` / `NEGA` / `BNE ?` / `INCX` / `?`
`OS byteval`	Operating system call: `SWI` / `DB byteval`
`OS8 farval`	Operating system call: `SWI` / `FAR farval` *(This one creates either a 3-byte or 4-byte sequence depending on `MMU` mode being off or on, respectively. Since v9.81)*
`OSW wordval`	Operating system call: `SWI` / `DW wordval`
`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`*
`ROLH`	Same as: `PSHH` / `ROL 1,ASP` / `PULH`
`RORH`	Same as: `PSHH` / `ROR 1,ASP` / `PULH`

Mnemonic/Syntax	Description
SEXA	Sign-extend A to XA, same as: CLRX / TSTA / BPL ? / COMX / ?
TAH	Same as: PSHA / PULH
THA	Same as: PSHH / PULA
THX	Same as: PSHH / PULX
TXH	Same as: PSHX / PULH
TPX	Same as: PSHA / TPA / TAX / PULA
TSTH	Same as: PSHH / TST 1,SP / PULH
TXP	Same as: PSHA / TXA / TAP / PULA
XGAX	Same as: PSHA / TXA / PULX
XGAH	Same as: PSHA / THA / PULH
XGHX	Same as: PSHH / TXH / PULX

Error, Warning, and Hint 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 code but the code produced may not run exactly as intended, or it may be inefficient. A program that has warnings may be totally correct and run as expected.

Hints inform the user about problems that do not prevent the assembler from producing correct code but there may be optimization opportunities.

Errors and warnings that begin with 'USER:' are generated by #ERROR and #WARNING directives, respectively. The source code author decides their meaning and importance. Hints generated by the #HINT directive do not begin with 'USER:', however.

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.

Errors

Error	Meaning
Invalid binary number	The string following the % sign is not made up of zeros and/or ones.
Binary number is longer than NN bits	A binary number may have no more than so many significant digits. Leading zeros are ignored.
"<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.
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.
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.
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.
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.
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.
Expression error	Something is wrong with the attempted expression, or an expression is altogether missing.
Invalid argument for DB directive	The value or expression supplied is not correct.

Error	Meaning
`Invalid argument for EQU/EXP/SET directive`	The value or expression supplied is not correct.
`Invalid first argument`	The first value or expression supplied is not correct.
`Invalid second argument`	The second value or expression supplied is not correct.
`Missing value between commas`	Found two (or more) commas without a value in between.
`Possibly duplicate symbol "<SYMBOL>"`	The symbol shown has already been defined. The word 'possibly' suggests that a symbol may have been truncated to 19 (or `#MaxLabel`) 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.
`Repeater value is invalid`	The repeater value (the :n part of the mnemonic) is a positive integer number.
`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.
`Undefined symbol " <SYMBOL>" or bad number`	The string shown is either a symbol that hasn't been defined at all, or it is a number that has some error, for example: `$ABCH` and `$FFFFF` are not valid hex numbers. The first contains an invalid character while the second is greater than 16 significant bits (`$FFFF`) in non-MMU mode.
`USER: <USER TEXT>`	This is a user generated error via the `#ERROR` directive.
`Comma not expected`	A comma was found in an unexpected position within the operand. Possibly using more arguments than required.
`Syntax error`	Some symbol is confusing the assembler. For example, `FCB #$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`). *Some other assemblers may have no problem with that but it is semantically incorrect.*
`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.
`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).
`ELSE without previous Ifxxx`	An `#ELSE` directive was encountered that does not match any unmatched `#IF` directive.
`ENDIF without previous Ifxxx`	An `#ENDIF` directive was encountered that does not match any unmatched `#IF` directive.
`Forward references not allowed`	The `#MEMORY`/`#VARIABLE` and other directives do not accept forward references.
`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 *(some other assembler do not use commas)* or forgotten to supply all the parts of the operand.
`Invalid expression(s) and/or comparator`	The expression or comparator used in the `#IF` directive is incorrect.
`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.
`Missing INCLUDE filename`	An `#INCLUDE` directive was supplied without any [path and] filename.
`Missing required first address`	A `#MEMORY`/`#VARIABLE` directive was encountered without any value or expression.
`Repeater value out of range (1-32767)`	The repeater value (the `:n` part of the mnemonic) must be from 1 to 32767.
`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.
`Symbol "<SYMBOL>" does not start with A..Z, . or _`	All symbols must start with one of the above characters. (File-local symbols start with a `?`)
`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.
`Too many include files. Maximum allowed is <NUMBER>`	The maximum number of INCLUDE files is <NUMBER> (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.
`Division by zero`	The expression used contains a division by zero after the / operator.
`MOD division by zero`	The expression used contains a division by zero after the \ operator.

Error	Meaning
"<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.
"<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.
Invalid argument for DW or FDB directive	The value or expression supplied is not correct.
Invalid argument for FAR directive	The value or expression supplied is not correct.
Invalid argument for ORG directive	The value or expression supplied is not correct.
Invalid argument for RMB or DS directive	The value or expression supplied is not correct.
Invalid argument for END directive	The value or expression supplied is not correct.
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

Warning	Meaning
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).
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).
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).
Violation of MEMORY directive at address $<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).
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).
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.
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.
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.
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!
RMB overlap at address $<VALUE> [Linear: $xxxxxx]	The variable of the shown line overlaps an already defined variable at the address shown. The warning appears at variables that cause the first and consequent overlaps but the problem could be with the original variable that was defined at this address. The assembler has no way of knowing your intentions!
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.
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 downsized to an 8-bit value, you will get a warning about this also).
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.

Warning	Meaning
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.
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.
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.
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.
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.
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.
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.
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.
USER: <USER TEXT>	This is a user generated warning via the #WARNING directive
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. You can prepend a ! to silence the warning.
Attempt to JUMP to page. Use CALL instead.	You are using a jump 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 JMP).
: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.
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.
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.
<"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).

Hints

Hint	Meaning
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.

Hint	Meaning
`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.
`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.
`Jumping to next instruction is needless`	You are using a jump 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.
`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.
`Shorter form wasn't used (forward reference?)`	There is a shorter instruction for this memory location but it wasn't used possible due to forward references.

Local Symbols

File-local Symbols

All symbols that begin with a question mark [?] are considered to be 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 same named symbols of the remaining participating files.

Advantages:

1. Symbols can be re-used in another INCLUDE file in a completely different way.
2. 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.

Procedure-local Symbols

All symbols that contain [@@] are considered to be local on a per-proc basis. Each proc can have its own locals that will not interfere with same named symbols of the remaining procs.

See the #PROC and PROC directives for details.

Advantages:

1. Symbols can be re-used in another proc in a completely different way.
2. Proc-local symbols are not visible outside the proc that contains them.

This last benefit makes it possible to write quite complex procedures while making only the subroutine entry label visible to the outside.

Macro-local Symbols

All symbols that contain [$$$] are considered to be local on a per-macro invocation basis. Each macro can have its own locals that will not interfere with same named symbols of the remaining macros or the different invocations of the same macro, even when calling itself.

See the macro directives for details.

Advantages:

1. Symbols can be re-used in another macro invocation in a completely different way.
2. Macro-local symbols are not visible outside the macro invocation that contains them.

This last benefit makes it possible to write quite complex macros while hiding all internal labels from outside access.

ASM8's Miscellaneous Features

Repeaters

Each mnemonic (but not macro calls) 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
                fcb:{COUNT+1} 0         ;Create a table of expression COUNT+1 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`).

Using `BSR`, `JSR`, and `CALL` instructions efficiently

Although the assembler does not restrict the programmer what instructions to use when calling subroutines, the following advice will help you keep your sanity in larger applications, so make it a habit to always do so in your programs.

Use `BSR` and `JSR` instructions exclusively for calling file local subroutines. (And this, assuming each file is only meant to reside in the same block of memory. In other words, a single file will be in the same logical memory space, whether that is non-paged, or paged memory for MMU equipped MCUs.)

File local subroutine names start with `?`. And, if the appropriate ASM8 option is turned on, you'll be hinted to change `JSR` to `BSR` when the distance is close enough, saving a byte for each change. Or use `!JSR` to avoid the hint. Make sure to terminate those subroutines with an `RTS` instruction.

Use `CALL` instructions to call global subroutines, i.e., subroutines with names not beginning with `?`. Important: Global subroutines can be defined in the same file they are used or some other file. What makes a subroutine global is its name, not where it is defined. Make sure to terminate those subroutines with `RTC` instructions.

If you follow this advice, NEVER assume a same file subroutine can be called with `BSR` or `JSR`. Look at the name instead. If it starts with `?` use `BSR` or `JSR`, otherwise use `CALL`. If you change a subroutine from global to local or vice-versa, make the necessary changes to all calls and don't forget to change the return instruction, also. Alternatively, add a wrapper subroutine so that your subroutine can effectively be called either way; locally for better efficiency, by using the `?` beginning name, and globally, otherwise. Two entry points, preferably named the same except for the leading `?` to mark the file-local version (as per assembler requirement). Like so:

```
                #spauto   :ab

Sub             proc                    ;global entry to be CALL-ed
                push                    ;save whatever caller's registers are used in our proc (if
any)
                #ais

                ...
                bsr       ?Sub
                ...

Done@@          ais       #:ais         ;de-allocate local stack variables (if any)
                pull                    ;restore caller's registers
                rtc
```

```
;*******************************************************************************

                    #spauto   2

?Sub                proc                          ;file-local entry to be BSR/JSR-ed
                    push                          ;save whatever caller's registers are used in our proc (if
any)
                    #ais

                    ...

Done@@              ais       #:ais               ;de-allocate local stack variables (if any)
                    pull                          ;restore caller's registers
                    rts
```

(Care must be taken when `?Sub` uses parent's stack variables. These must be copied from the global `Sub` before the `BSR` to `?Sub` and the results copied back to the parent stack after the `BSR` to `?Sub`.)

The `CALL` instruction is normally only available in MMU equipped variants of 9S08 MCUs. But, by default configuration, the assembler will automatically convert all `CALL` and `RTC` instructions to `JSR` and `RTS` ones, respectively but only when in `#NoMMU` (`-mmu-` command-line) mode.

It will also not issue hints when the target gets too close for a `BSR` instruction as you wouldn't want to change these to `BSR` so they continue working when the code (e.g. a general library) is also used in MMU mode in some other application.

This makes your code 100% compatible with both MMU and non-MMU variants. All you need is to use the `#MMU` or `#NoMMU` directive, respectively. (Normally, this would be done inside the MCU specific include file, only once for each MCU.)

If using ASM8's automatic stack offset adjustments (highly recommended), your global subroutines include the `#spauto` directive as shown in the previous example:

The `:ab` will be converted to 3 when in MMU mode, and 2 when out of MMU mode.

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 one hundred 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 ?
          #Hint     *****************************************
          #Hint     * Choice of run-time conditional symbols
          #Hint     *****************************************
          #Hint     * DEBUG: Turns on debugging code
          #Hint     * QE128: Target is MC9S08QE128
          #Hint     * GP   : Target is MC68HC908GP32
          #Hint     *****************************************
          #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, but be consistent or you'll lose the benefit of this 'trick'.)

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

                    #ROM      ROM                 ;(switch to ROM segment and ORG it at ROM)

Start               proc
                    rsp                           ;the program begins here
                    ...                           ;(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. (You can use either `:` or `=` to assign a value to the symbol.)

As another example, you could declare an array where you override the default dimension during assembly. No need to edit the source. The same source could produce different versions.

```
;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           ;status for each object
.object             rmb       ARRAYSIZE*2         ;pointer to each object
                    ...

                    #ROM

Start               proc                          ;the program begins here
                    rsp
                    ...                           ;(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 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 de-select 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               proc                          ;the program begins here
                    rsp
                    ...                           ;(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.

My own source code style and naming conventions

Synopsis:

- Tabs are never used.
- End of line spaces are removed (automatically by the editor or a separate utility as required).
- End-of-line is Linux style (LF) even under Windows.
- Labels begin in column 1 (assembler requirement).
- Non-conditional directives begin in column 21.
- Conditional directives outside a code block begin in column 1.
- Conditional directives inside a code block begin in column 11.
- Each nested conditional directive is indented by 2 spaces from its parent.
- Instruction mnemonics and most macro calls begin in column 21.
- Instruction operands begin in column 31.
- End of line comments begin in column 51 unless a long operand pushes them further to the right, in which case at least one space separates the comment from the operand.
- All comments begin with semi-colon (the assembler does not enforce this).
- Stand-alone comments begin in column 1.
- In-line proc comments begin in column 11 or follow a dotted line (;---) that begins in column 11 and runs to column 49, leaving column 50 blank, then a semi-colon in column 51 followed by the comment text.
- Dot-ending names (XXX.) indicate a unique bit by position number (0 thru 7).
- Underscore-ending names (XXX_) indicate a bit mask (possibly including multiple bits).
- Underscore surrounded names (_XXX_) indicate special system symbols such as CCR offsets, etc.
- All constants are uppercase and use underscores to separate words as needed (MY_CONSTANT).
- All variables are lowercase and use underscores to separate words as needed (e.g., my_var).
- All pointer variables begin with a point/dot (.var).
- Code labels use CamelCase.
- File-local labels begin with ? (a question mark). This is an assembler requirement.
- Proc-local labels end with @@ (although the assembler allows @@ to be anywhere past the first character of the label).
- Macro-local labels end with $$$ (although the assembler allows $$$ to be anywhere past the first character of the label).
- Subroutines always begin with a proc pseudo-op to mark the beginning of the subroutine. This allows proc-local symbols to be used inside the subroutine without worrying about name collisions with other subroutines’ symbols.
- Subroutines normally use #spauto mode with optional :ab offset when the proc may reference caller-level stack variables.
- Local (stack) variables are defined near the beginning of the subroutine either directly (with AIS #-nn) or with the use of the local macro which follow an #AIS directive. These variables are de-allocated using the AIS #:AIS instruction.
- Each subroutine is separated from the previous one with an 80-column semi-colon beginning comment line full of asterisks (i.e., ;***...).
- File-local subroutine names begin with ? as required by the assembler to keep the name local to that file.
- File-local subroutines end with an RTS instruction. They are called with either JSR or BSR instructions.
- Global subroutines end with an RTC instruction. They are called exclusively with the CALL instruction (for MMU compatibility).
- Very small (few lines) and obvious purpose subroutines usually have no spaces between instructions.

- Regular subroutines separate each block of related assembly language instructions with a single blank line, or a dotted comment line (see earlier).
- Segments are used to separate object code into distinct areas. Segment `#RAM` is used for zero-page RAM variables that may need `B[R]SET` or `B[R]CLR` instruction access. Segment `#XRAM` is used for all other variables. Segment `#ROM` is used for normal code. Segment `#VECTORS` is used for vector definitions.
- File inclusion is done primarily with the `#Uses` directive. `#Include` is only used when the same file must be included more than once (where `#Uses` would fail by design), or when we need to be sure the inclusion point is at the `#Include` directive (since a `#Uses` may have already included the file at some earlier point).
- All file inclusion paths are relative and depend on the assembler's default include path search. Specifically, they are first relative to the current file being assembled, then to the main file of the current assembly, and finally to the root file (filename `_asm_`) if one exists.
- Ad-hoc `?` macros are used extensively to simplify repeated sequential coding of complicated expressions or coding patterns.
- Hungarian notation is never used.

Some of the above points in a bit more detail…

`_MY_OFFSETS_` are constants (see above) surrounded by single extra underscores.

`MyRoutines` are CamelCase names. They may also use underscores mostly to separate module from function name, as in `InitializeSCI` which could also be written as `Initialize_SCI` or `SCI_Initialize` (preferred).

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

Proc local symbols follow the above rules but always end with `@@`, example: `Loop@@` by requirement of the assembler. (Note: The assembler requirement is only for the presence of `@@` anywhere after the first character and not that they appear at the end of the label as in my own convention.)

Macro local symbols follow the above rules but always end with `$$$`, example `Loop$$$` by requirement of the assembler. (Note: The assembler requirement is only for the presence of `$$$` anywhere after the first character and not that they appear at the end of the label as in my own convention.)

Comments always start with a semicolon. Macro comments that do not need expansion start with a double semicolon, and are not saved with the macro.

Labels are defined starting in column 1 (assembler requirement) and use the default maximum label length of 19 (including expansion of `@@` or `$$$`) so they remain compatible with *P&E Micro* map files.

Mnemonics and pseudo mnemonics (e.g., `ORG`) start in column 21 and are always written in lowercase. Exception to the lowercase convention is when an instruction is meant to do something other than the obvious, e.g., `RTS` when used to `JMP` to a previously stacked address will be written in uppercase to signify the special use.

Macro calls normally start in column 21 just like regular mnemonics but they may also start in column 1 if, and only if, they are invoked as `@macro` (leading `@`). This is useful when there are long operands to make it easier to view without any/much horizontal scrolling of the editor window.

Operands start in column 31 unless a long repeater expression or macro name pushes them further to the right. For macro calls that start earlier than column 21, operands may start anywhere, usually after a single space from the macro name.

Comments start in column 51 unless a long operation/operand combination pushes them further to the right, separated by at least one (preferably two) space character(s).

Procs are separated with an 80-char comment line full of asterisks that always begins with a semicolon. (Most editors allow shortcut macros that quickly produce such template code.)

A non-trivial `proc` uses a header with `Purpose`, `Input`, `Output`, and `Note(s)` that describes the purpose of the proc and the calling interface. Something like:

```
;*******************************************************************************
; Purpose:
; Input  : HX ->
;        : A =
; Output : Carry Clear on success, Carry Set on Error
;        : HX ->
;        : A =
; Note(s):
```

All procs use automatic SP offset adjustment with the `#spauto` assembler directive immediately preceding the `proc` directive and separated from it with a blank line.

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

Each proc begins with the proc name and the assembler directive `proc`. An `endp` directive at the end of the `proc` is normally not used, unless we specifically want to embed a `proc` inside another, usually much larger `proc`, for proximity reasons (e.g., frequent `BSR` use optimization), in which case we need to protect the parent proc's label locality.

Each proc protects all caller registers except for CCR (with few exceptions that need to also protect CCR), and those registers that pass out return values. Usually the `PUSH` instruction is used right after `proc` and the `PULL` right before `RTS` or `RTC`. If RegA is used to return a value then either we replace `PUSH` and `PULL` with `PSHHX` and `PULHX` or we name the return register and save the result to it, like so:

```
                    #spauto

Sub                 proc
                    psha      ans@@
                    pshhx
                    #ais

                    ...
                    sta       ans@@,sp            ;save result for caller
                    ...

Done@@              ais       #:ais
                    pull
                    rtc
```

Procs are written in a modular (structured) way with entry, main body, and exit sections. (There are exceptions when clarity of purpose is more evident, otherwise.)

Proc-local labels are used in a consistent manner where possible. So, a `proc` with a single loop will have it start at label `Loop@@` and continue at the label `Cont@@`.

A `proc`'s exit section begins at label `Done@@` where any exit requirements are enforced. This usually includes releasing temporary stack variables with the `AIS #:AIS` instruction that automatically releases as many bytes as were allocated by either `AIS` or `PSHx` instructions at the entry section.

Each `proc` may return (in addition to any registers), a success or failure status in the CCR[C] register bit. Success is always CCR[C] = 0 which is achieved with a single `CLC` instruction. Failure, on the other hand, requires a `SEC` instruction. Note this is different from C language convention where a zero is considered false. Here, a zero Carry flag is success (or true in true/false determining `proc`).

The body section should release its own stack as it is used.

Note: My own code and/or examples may not yet adhere to all of the above, as most of it was written long before these conventions were put into action.

But, each time some code gets updated, I try to also update the coding style to conform to my most current conventions. And, the conventions are subject to improve (in my subjective view) with experience.

Each instruction (with the exception of trivial operations) is documented in the comments area with appropriate comments that do not repeat the instruction but explain what happens and/or for what purpose.

Following the above conventions has helped me produce consistently bug-free code (within reasonable expectations), and easy to read logic.

Linux/Win32 version addendum

The DOS, Win, and Linux versions are practically identical. Where there are differences, these are noted.

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

Standard error redirection works for Windows, and all messages, hints, warnings, and errors are redirected. For Linux *all output* (not just the errors) is redirected, but this may not be in a very readable format as it may include the line counters during Pass1 and Pass2. To have proper ERR files created, please use only the `-E` option.

Beginning with v1.20, the DOS/Win32 versions allow wildcards on the command line for matching multiple assembly language filenames. The Linux version uses standard Linux syntax for multiple filenames. For Win32/Linux case, filenames are not limited to the DOS 8.3 format. The `#INCLUDE` directives within the source code may also specify long filenames.
Currently, spaces within long paths or filenames are not supported.

For all versions, you must keep the included `asm8.cfg` in the same directory as the asm8[.exe] binary (for example, `~/bin` for Linux or `C:\Utils` for DOS/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. Note: If you happen to rename the executable to whatever other name, you MUST also rename the .cfg file to the same base name, or it won't be found.

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[.exe] binary is, and anytime a local `asm8.cfg` isn’t found, the global one will be used instead.

Another difference imposed by Linux 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.

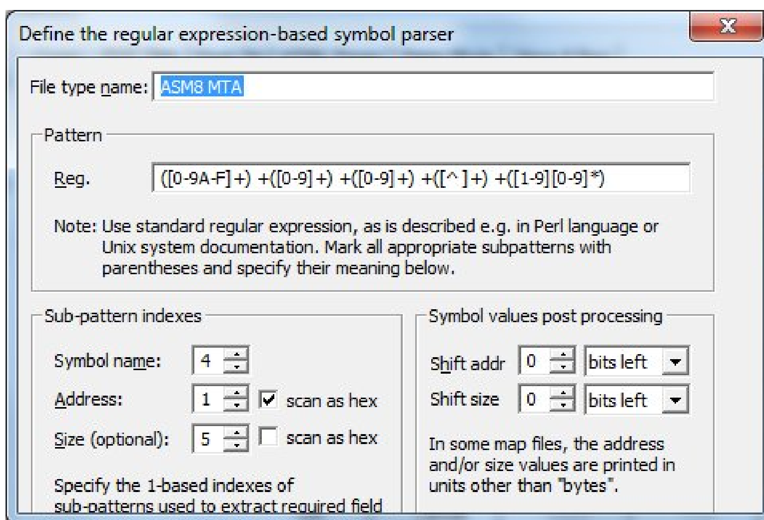
These are pretty much the only differences in behavior.

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

Setting up FreeMASTER (from Freescale/NXP) for use with ASM8

IMPORTANT: You must use the `-MTA` option for generation of ASM8 map files. This will create ASCII map files that can be parsed by a regular expression text parser. (The default *P&E Micro* map files cannot be used for this purpose.)

In FreeMASTER application, go to `Project/Options/MAP Files` option and define a RegExp text parser as the File Format. Then fill out the screen that appears as follows: `([0-9A-F]+) +([0-9]+) +([0-9]+) +([^ ]+) +([1-9][0-9]*)`



FreeMASTER

ASM8 v12.30, February 13, 2022, Copyright (c) 2001-2022 by Tony G. Papadimitriou (*email: tonyp@acm.org*)