Topics

Examples

List of Tables

Notes

4 Assembler Directives

Assembler directives supply program data and control the assembly process. Assembler directives enable you to do the following:

- Assemble code and data into specified sections
- Reserve space in memory for uninitialized variables
- Control the appearance of listings
- Initialize memory
- Assemble conditional blocks
- Define global variables
- Specify libraries that the assembler can obtain macros from
- Generate symbolic debugging information

4.1 Directives Summary

The table summarizes the assembler directives. *Note that all source statements that contain a directive may have a label and a comment*. To improve readability, they are not shown as part of the directive syntax.

Directives That Define Sections	
Mnemonic and Syntax	*Description*
.bss symbol [, size in bytes, address]	Reserve size bytes in the .bss (uninitialized data) section
.data [*address*]	Assemble into the .data (initialized data) section
.sect *"section name"* [*, address*]	Assemble into a named (initialized) section
.text *[address]*	Assemble into the .text (executable code) section
symbol **.usect** *"section name", size in bytes [, address]*	Reserve *size* bytes in a named (uninitialized) section

Directives That Initialize Constants (Data and Memory)	
Mnemonic and Syntax	*Description*
.byte $value_1$ [*, ... ,* $value_n$]	Initialize one or more successive bytes in the current section
.double *floating point value*	Initialize a 48-bit MSP430 floating-point constant
.field *value* [*, size in bits*]	Initialize a variable–length field
.float *floating point value*	Initialize a 32-bit, MSP430 floating–point constant
.space *size in bytes*	Reserve *size* bytes in the current section; note that a label points to the beginning of the reserved space
.string $"string_1"$ [*, ... ,* $"string_n"$]	Initialize one or more text strings
.word $value_1$ [*, ... ,* $value_n$]	Initialize one or more 16–bit integers

Table 4.1: Assembler Directives Summary

Directives That Align the Section Counter (SPC)	
Mnemonic and Syntax	*Description*
.align	Align the SPC on a byte boundary
.even	Align the SPC on a word boundary

Directives That Format the Output Listing	
Mnemonic and Syntax	*Description*
.fclist	Allow false conditional code block listing (default)
.fcnolist	Inhibit false conditional code block listing
.length *page length*	Set the page length of the source listing
.list	Restart the source listing
.mlist	Allow macro listings and loop blocks (default)
.mnolist	Inhibit macro listings and loop blocks
.nolist	Stop the source listing
.option *{A|B|F|M|T|W|X}*	Select output listing options
.page	Eject a page in the source listing
.sslist	Allow expanded substitution symbol listing
.ssnolist	Inhibit expanded substitution symbol listing (default)
.title *“string"*	Print a title in the listing page heading
.width *page width*	Set the page width of the source listing

Table 4.1: Assembler Directives Summary (Continued)

Directives That Reference Other Files	
Mnemonic and Syntax	*Description*
.copy [“]*filename*[*"*]	Include source statements from another file
.def $symbol_1$ [*, ... ,* $symbol_n$]	Identify one or more symbols that are defined in the current module and used in other modules
.global $symbol_1$ [*, ... ,* $symbol_n$]	Identify one or more global (external) symbols
.include [“]*filename*[*"*]	Include source statements from another file
.mlib [“]*filename*[*"*]	Define macro library
.ref $symbol_1$ [*, ... ,* $symbol_n$]	Identify one or more symbols that are used in the current module but defined in another module

Conditional Assembly Directives	
Mnemonic and Syntax	*Description*
.break [*well–defined expression*]	Optional repeatable block assembly
.if *well–defined expression*	Begin conditional assembly
.else	Optional conditional assembly
.elseif well–defined expression	Optional conditional assembly
.endif	End conditional assembly
.endloop	End repeatable block assembly
.loop [*well–defined expression*]	Begin repeatable block assembly

Table 4.1: Assembler Directives Summary (Continued)

Assembly–Time Symbols	
Mnemonic and Syntax	*Description*
.asg [“] *character string* [*”*], *substitution*	Assign a character string to a substitution symbol
.endstruct	End structure definition
.equ	Equate a value with a symbol
.eval *well–defined expression, sub–stitution symbol*	Perform arithmetic on numeric substitution symbols
.newblock	Undefine local labels
.set	Equate a value with a symbol
.struct	Begin structure definition
.tag	Assign structure attributes to a label

Miscellaneous Directives	
Mnemonic and Syntax	*Description*
.emsg string	Send user–defined error messages to the output device
.end	Program end
.label *“symbol"*	Define a load address label
.mmsg *string*	Send user–defined messages to the output device
.setsect *“section name",addr*	Produced by absolute lister, See Chapter 9
symbol **.setsym** *addr*	Produced by absolute lister, See Chapter 9
.wmsg *string*	Send user–defined warning messages to the output device

Table 4.1: Assembler Directives Summary (Concluded)

4.2 Directives That Define Sections

Five directives associate the various portions of an assembly language program with the appropriate sections:

- **.bss** reserves space in the .bss section for uninitialized variables.
- **.data** identifies portions of code in the .data section. The .data section usually contains initialized data.
- **.sect** defines initialized named sections and associates subsequent code or data with that section. A section defined with .sect can contain code or data.
- **.text** identifies portions of code in the .text section. The .text section usually contains executable code.
- **.usect** reserves space in an uninitialized named section. The .usect directive is similar to the .bss directive, but it allows you to reserve space separately from the .bss section.

The output listing on the next page shows how you can use sections directives to associate code and data with the proper sections. Column 1 shows line numbers, and column 2 shows the SPC values. (Each section has its own program counter, or SPC.) When code is first placed in a section, its SPC equals 0. When you resume assembling into a section, its SPC resumes counting as if there had been no intervening code.

After the code in is assembled, the sections contain:

.text	Initializes bytes with the values 1, 2, 3, 4, 5, 6, 7, and 8
.data	Initializes bytes with the values 9, 10, 11, 12, 13, 14, 15, and 16
var_defs	Initializes bytes with the values 17 and 18
.bss	Reserves 19 bytes
xy	Reserves 20 bytes

Note that the .bss and .usect directives do not end the current section or begin new sections; they reserve the specified amount of space, and then the assembler resumes assembling code or data into the current section.

```
 1                      ;**********************************
 2                      ;* Start assembling into .text    *
 3                      ;**********************************
 4 0000                       .text
 5 0000   01                  .byte 1,2
   0001   02
 6 0002   03                  .byte 3,4
   0003   04
 7
 8                      ;**********************************
 9                      ;* Start assembling into .data    *
10                      ;**********************************
11 0000                       .data
12 0000   09                 .byte 9,10
   0001   0a
13 0002   0b                 .byte 11,12
   0003   0c
14
15                      ;**********************************
16                      ;* Start assembling into named    *
17                      ;* section, var_defs              *
18                      ;**********************************
19 0000                       .sect "var_defs"
20 0000   11                 .byte 17,18
   0001   12
21
22                      ;**********************************
23                      ;* Resume assembling into .data   *
24                      ;**********************************
25 0004                       .data
26 0004   0d                 .byte 13,14
   0005   0e
27 0000                       .bss sym,19    ; reserve space in .bss
28 0006   0f                 .byte 15,16    ; still in .data
   0007   10
29
30                      ;**********************************
31                      ;* Resume assembling into .text   *
32                      ;**********************************
33 0004                       .text
34 0004   05                 .byte 5,6
   0005   06
35 0000                 usym  .usect "xy",20 ; reserve space in xy
36 0006   07                 .byte 7,8      ; still in .text
   0007   08
```

Example 4.1: Sections Directives

4.3 Directives That Initialize Memory

Several directives initialize memory:

- **.byte** places one or more 8–bit values into consecutive bytes of the current section.
- **.word** places one or more 16–bit values into consecutive bytes in the current section.
- **.string** places 8–bit characters from one or more character strings into the current section. This directive is identical to .byte.
- **.float** calculates a (32–bit) MSP430 floating–point representation of a single precision floating–point value and stores it in four consecutive bytes in the current section.
- **.double** calculates a (48 bit) MSP430 floating-point representation of a double precision floating-point value and stores it in six consecutive bytes in the current section.

  The following code has been assembled for the example, that compares the .byte, .float, .word, and .string directives:

```
1 0000   aa                    .byte 0AAh, 0BBh
  0001   bb
2 0002   1234                  .word 01234h
3 0004   68                    .string "help"
  0005   65
  0006   6c
  0007   70
4 0008   81490fdb              .float  3.141592654
5 000C   81490fdaa292          .double 3.141592654
```

Note: How the .byte, .word, .string, .float, .double and .field Directives Function in a .struct/.endstruct Sequence

The .byte, .word, .string, .float, .double and .field directives do not initalize memory when they are part of a .struct/.endstruct sequence; rather, they define a member´s size.

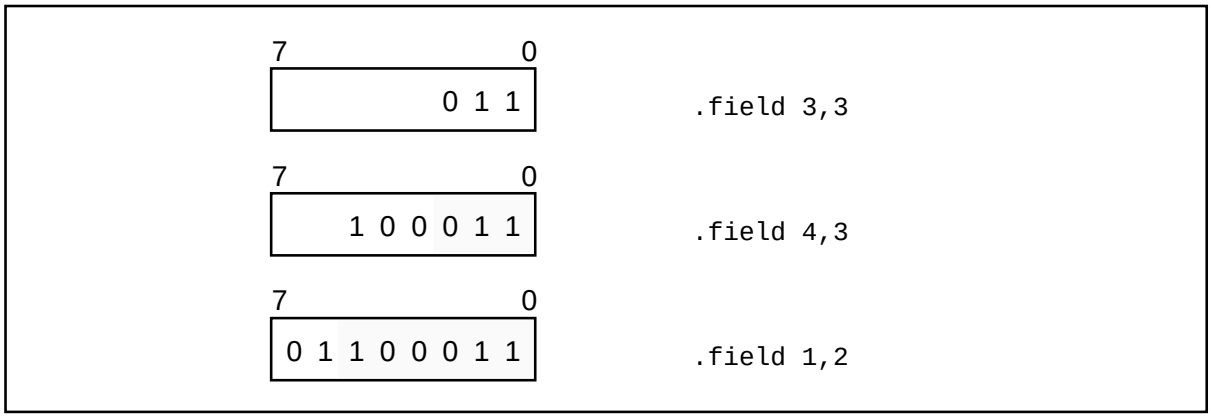
BYTE	CODE	
0,1	7 0 7 0 A A B B	.byte 0AAh, 0BBh
2,3	7 0 7 0 1 2 3 4	.word 01234h
4,5	7 0 7 0 6 8 6 5	.string "help"
6,7	6 C 7 0	
8,9	7 0 7 0 8 1 4 9	.float 3.141592654
10,11	0 F D B	
12,13	7 0 7 0 8 1 4 9	.double 3.141592654
14,15	0 F D A	
16,17	A 2 9 2	

Example 4.2: Initialization Directives

- The **.field** directive places a single value into a specified number of bits in the current byte. With a field, you can pack multiple fields into a single byte; the assembler does not increment the SPC until a byte is filled.

  The next example shows how fields are packed into a byte. For this example, assume the following code has been assembled; note that the SPC doesn't change. (The fields are packed into the same byte.)

```
1     0000  03       .field   3,3
2     0000  23       .field   4,3
3     0000  63       .field   1,2
```



Example 4.3: The .field Directive

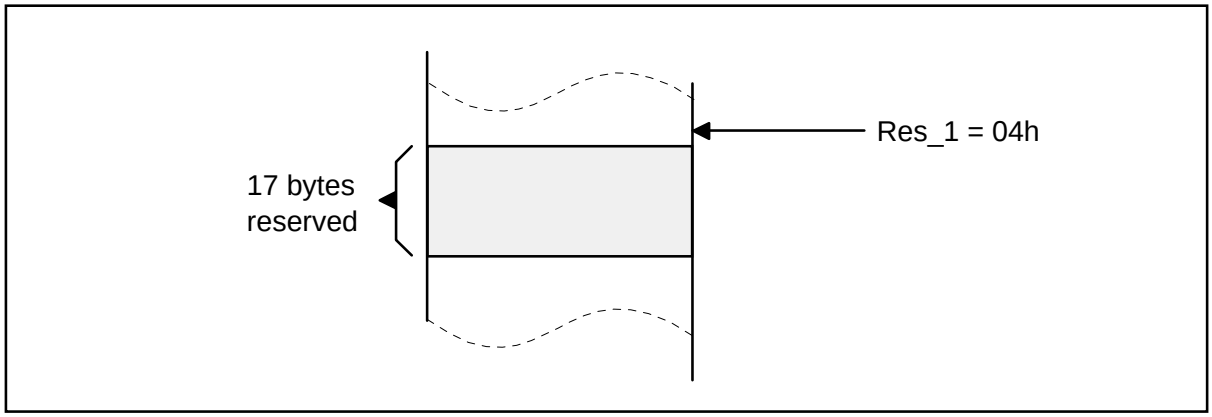
- The **.space** directive reserves a specified number of bytes in the current section. The assembler fills these reserved bytes with 0s.

  When you use a label with .space, it points to the *first* byte of the reserved block.

  The following code has been assembled for the example of the .space directive:

```
45 0000   0100                    .word 100h,200h
   0002   0200
46 0004                   Res_1:   .space 17
47 0016   000f                     .word 15
```

  Res_1 points to the first byte in the space reserved by .space.

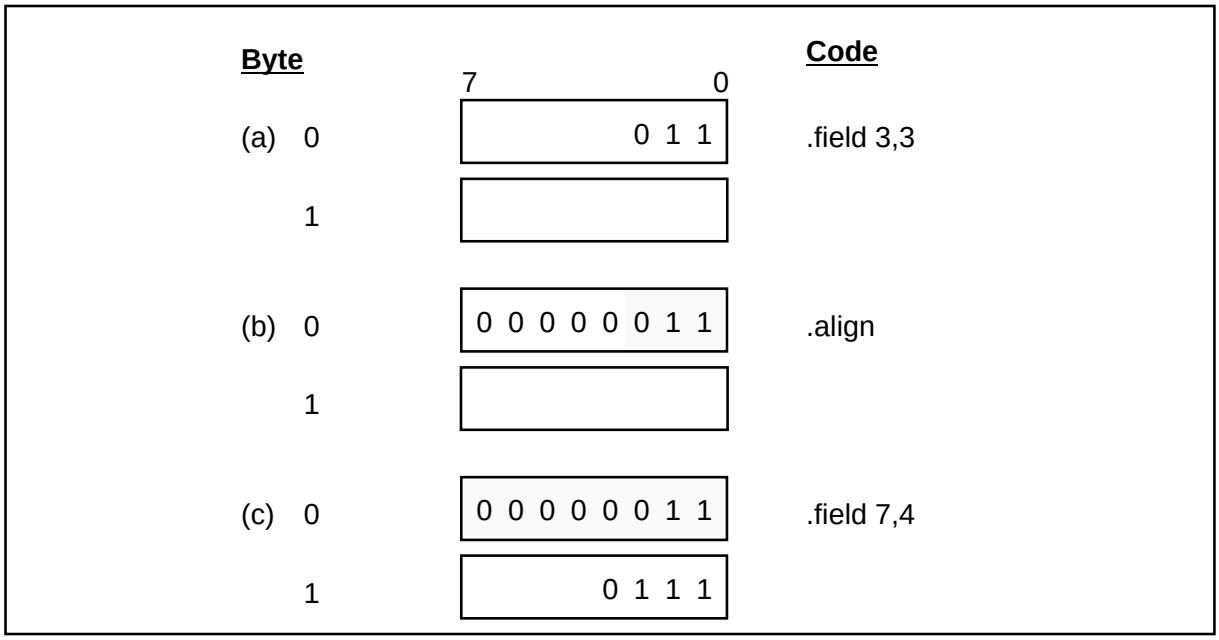


Example 4.4: The .space Directive

4.4 Directives That Align the Section Program Counter

- The .align directive aligns the SPC at the next byte boundary. This directive is useful with the .field directive when you do not wish to pack two adjacent fields in the same byte. The following code has been assembled for the example:

```
1 0000   03                    .field   3,3
2                              .align
3 0001   07                    .field   7,4
4
```



Example 4.5: The .align Directive

- The .even directive aligns the SPC at the next word boundary. It can be used to align bytes or strings on word boundaries:

```
1 0000   41                    .string "ABC"
  0001   42
  0002   43
2                              .even
3 0004   58                    .string "XYZ"
  0005   59
  0006   5a
4
```

4.5 Directives That Format the Output Listing

The following directives format the listing file:

- The source code contains a listing of false conditional blocks that do not generate code. The **.fclist** and **.fcnolist** directives turn this listing on and off. You can use the .fclist directive to list false conditional blocks exactly as they appear in the source code. You can use the .fcnolist directive to list only the conditional blocks that are actually assembled.
- The **.length** directive controls the page length of the listing file. You can use this directive to adjust listings for various output devices.
- The **.width** directive controls the page width of the listing file. You can use this directive to adjust listings for various output devices.
- The **.list** and **.nolist** directives turn the output listing on and off. You can use the .nolist directive to prevent the assembler from printing selected source statements in the listing file. Use the .list directive to turn the listing back on.
- The **.mlist** and **.mnolist** directives allow and inhibit macro expansion and loop block listings. You can use the .mlist directive to print all macro expansions and loop blocks to the listing.
- The **.option** directive controls several features in the listing file. This directive has several operands:

  A Turns on all listing (overrides all other directives and options).

  B Limits the listing of .byte directives to one line.

  F Resets the B, W, M, and T directives.

  M Turns off macro expansions in the listing.

  T Limits the listing of .string directives to one line.

  W Limits the listing of .word directives to one line.

  X Produces a cross–reference listing of symbols. (You can also obtain a cross–reference listing by invoking the assembler with the -x option.)
- The **.page** directive causes a page eject in the output listing.
- The **.sslist** and **.ssnolist** directives allow and inhibit substitution symbol expansion listing. These directives are useful for debugging substitution symbols outside of macros.
- The **.title** directive supplies a title that the assembler prints at the top of each page.

4.6 Directives That Reference Other Files

These directives supply information for or about other files:

- The **.copy** and **.include** directives tell the assembler to begin reading source statements from another file. When the assembler finishes reading the source statements in the copy/include file, it resumes reading source statements from the current file. The statements read from a copied file are printed in the listing file; the statements read from an included file are *not* printed in the listing file.
- The **.global** directive declares a symbol to be external so that it is available to other modules at link time. This directive does double duty, acting as a .def for defined symbols and as a .ref for undefined symbols. Note that the linker will resolve an undefined global symbol only if it is used in the program. The .global directive declares a symbol as 16 bits.
- The **.def** directive identifies a symbol that is defined in the current module and can be used by other modules. The assembler puts the symbol in the symbol table.
- The **.ref** directive identifies a symbol that is used in the current module but defined in another module. The assembler marks the symbol as an undefined external symbol and puts it in the object symbol table so that the linker can resolve its definition.
- The **.mlib** directive supplies the assembler with the name of an archive library that contains macro definitions. When the assembler encounters a macro that is not defined in the current module, it searches for it in the macro library specified with .mlib.

4.7 Conditional Assembly Directives

Conditional assembly directives enable you to instruct the assembler to assemble certain sections of code according to a true or false evaluation of an expression. Two sets of directives allow you to assemble conditional blocks of code:

- The **.if/.elseif/.else/.endif** directives tell the assembler to conditionally assemble a block of code according to the evaluation of an expression.

.if *expression*	Marks the beginning of a conditional block and assembles code if the .if condition is true.
.elseif *expression*	Marks a block of code to be assembled if .if is false and .elseif is true.
.else	Marks a block of code to be assembled if .if is false.
.endif	Marks the end of a conditional block and terminates the block.

- The **.loop/.break/.endloop** directives tell the assembler to repeatedly assemble a block of code according to the evaluation of an expression.

.loop *expression*	Marks the beginning a repeatable block of code.
.break *expression*	Continue to repeatedly assemble when the .break expression is false. Go to code immediately after .endloop if expression is true.
.endloop	Marks the end of a repeatable block.

The assembler supports several relational operators that are especially useful for conditional expressions.

4.8 Assembly–Time Symbol Directives

These directives equate meaningful symbol names to constant values or strings.

- The **.set** and **.equ** directives set a constant value to a symbol. The symbol is stored in the symbol table and cannot be redefined; for example:

```
bval  .set   0100h
      .word  bval, bval*2, bval+12
      br     bval
```

  Note that the .set and .equ directives produce no object code.

- The **.struct/.endstruct** directives set up C–like structure definitions, and the **.tag** directive assigns the C–like structure characteristics to a label.

  The .struct./endstruct directives enable you to set up a C–like structure definition so that similar elements can be grouped together. Element offset calculation is then left up to the assembler. The .struct/.endstruct directives do not allocate memory. They simply create a symbolic template that can be used repeatedly.

  The .tag directive assigns structure characteristics to a label. This simplifies the symbolic representation and also provides the ability to define structures that contain other structures.The .tag directive does not allocate memory, and the structure tag (stag) must be defined before it is used.

```
type  .struct                ; structure tag definition
x     .byte
y     .byte
t_len .endstruct

coord .tag   type            ; declare coord (coordinate)
      .usect coord,t_len     ;actual?memory?allocation
      add    coord.y,R4
```

- The **.asg** directive assigns a character string to a substitution symbol. The value is stored in the substitution symbol table. When the assembler encounters a substitution symbol, it replaces the symbol with its character string value. Substitution symbols can be redefined.

```
      .asg  "10, 20, 30, 40", coefficients

      .byte coefficients
```

- The **.eval** directive evaluates an expression, translates the results into a character string, and assigns the character string to a substitution symbol. This directive is most useful for manipulating counters; for example:

```
      .asg   1 , x
      .loop
      .byte  x*10h
      .break   x = 4
      .eval  x+1, x
      .endloop
```

4.9 Miscellaneous Directives

This section discusses miscellaneous directives.

- The **.setsect** directive is generated by the absolute lister. It specifies an absolute starting address for a section name so that the assembler can generate an absolute listing.
- The **.setsym** directive is generated by the absolute lister. It specifies an absolute address for a global symbol. This allows the assembler to generate an absolute listing.
- The **.end** directive terminates assembly. It should be the last source statement of a program. This directive has the same effect as an end–of–file.
- The **.label** directive defines a special symbol that refers to the loadtime address rather than the runtime address within the current section.

These three directives enable you to define your own error and warning messages:

- The **.emsg** directive sends error messages to the standard output device. The .emsg directive generates errors in the way the assembler does, incrementing the error count and preventing the assembler from producing an object file.
- The **.wmsg** directive sends warning messages to the standard output device. The .wmsg directive functions in the same way the .emsg directive does, but increments the warning count and does not prevent the assembler from producing an object file.
- The **.mmsg** directive sends assembly–time messages to the standard output device. The .mmsg directive functions in the same way the .emsg and .wmsg directives do, but does not set the error count or the warning count and does not prevent the assembler from producing an object file.

4.10 Directives Reference

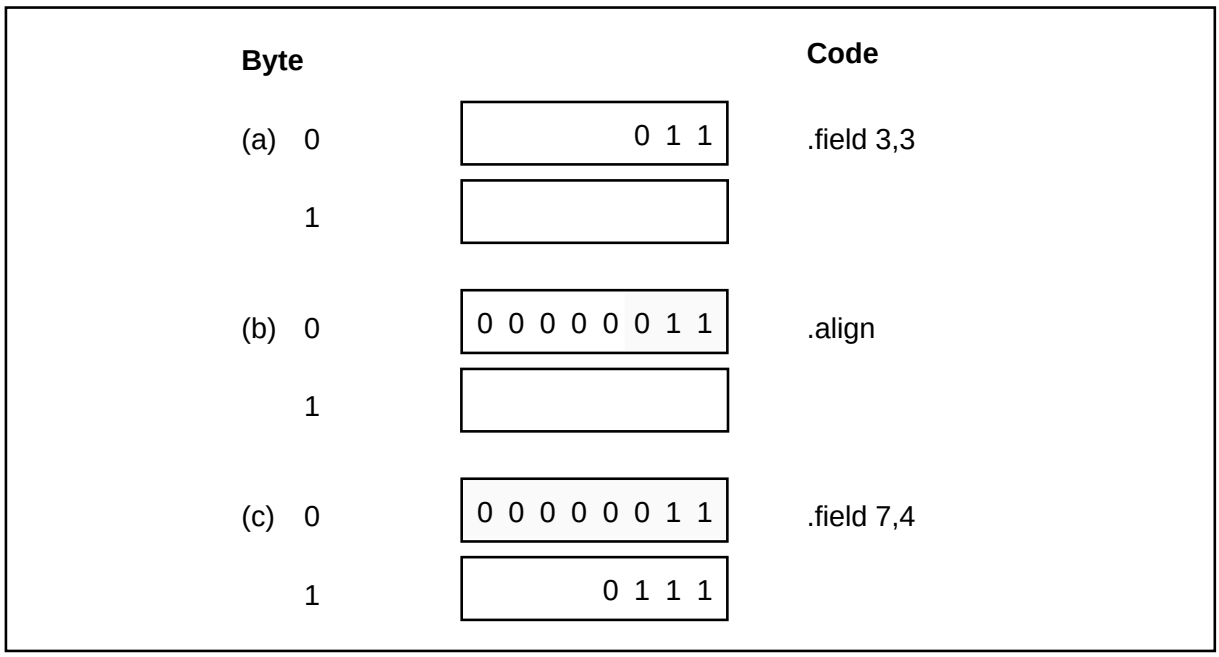
The remainder of this chapter is a reference. Generally, the directives are organized alphabetically, one directive per page; related directives (such as .if/.else/.endif), however, are presented together on one page. Here's an alphabetical table of contents for the directives reference:

Syntax **.align**

Description The .align directive aligns the section program counter (SPC) on the next byte boundary. This directive is useful with the .field directive when you do not wish to pack two adjacent fields in the same byte.

Example This example shows the creation of two fields that would normally be packed within the same byte. The .align directive forces these fields into separate bytes.

```
1 0000   03                    .field   3,3
2                              .align
3 0001   07                    .field   7,4
4
```



Example 4.6: The .align Directive

Syntax **.asg** [*"*] *character string*[*"*]*, substitution symbol*

.eval *well–defined expression, substitution symbol*

Description The **.asg** directive assigns character strings to substitution symbols; substitution symbols are stored in the substitution symbol table.

The .asg directive can be used in many of the same ways as the .set directive, but while .set assigns a constant value (cannot be redefined) to a symbol, .asg assigns a character string (can be redefined) to a substitution symbol.

- The assembler assigns the *character string* to the substitution symbol. The quotation marks are optional. If there are no quotation marks, the assembler reads characters up to the first comma and removes leading and trailing blanks. In either case, a character string is read and assigned to the substitution symbol.
- The *substitution symbol* is a required parameter that must be a valid symbol name. The substitution symbol may be 32 characters long and must begin with a letter. Remaining characters of the symbol can be a combination of alphanumeric characters, underscores, and dollar signs.

The **.eval** directive performs arithmetic operations on substitution symbols. This directive evaluates the expression and assigns the **string value** of the result to the substitution symbol. The .eval directive is especially useful as a counter in .loop/.endloop blocks.

Example This example shows how .asg and .eval can be used.

```
                1                   .sslist; show expanded sub.syms
                2                ;*
                3                ;* .asg/.eval example
                4                ;*
                5                          .asg    &, AND
                6                          .asg    R11, FP
                7
                8 0000 503b000c            add     #32 AND 44, FP
#                                          add     #32 & 44, R11
                9
               10                          .asg    0, x
               11                          .loop   5
               12                          .eval   x+1, x
               13                          .word   x
               14                          .endloop
1                                          .eval   x+1, x
#                                          .eval   0+1, x
1                  0004 0001               .word   x
#                                          .word   1
1                                          .eval   x+1, x
#                                          .eval   1+1, x
1                  0006 0002               .word   x
#                                          .word   2
1                                          .eval   x+1, x
#                                          .eval   2+1, x
1                  0008 0003               .word   x
#                                          .word   3
1                                          .eval   x+1, x
#                                          .eval   3+1, x
1                  000a 0004               .word   x
#                                          .word   4
1                                          .eval   x+1, x
#                                          .eval   4+1, x
1                  000c 0005               .word   x
#                                          .word   5
```

Syntax **.bss** *name* [*, size in bytes, address*]

Description The .bss directive reserves space in the .bss section for variables. Use this directive to allocate space into RAM.

- The *name* is a required parameter. It defines a symbol that points to the first location reserved by the directive.
- The *size* is an optional parameter. It is a well–defined, absolute expression that specifies the number of bytes that are allocated. The default size for this directive is 1 byte.
- The *address* is an optional parameter that specifies a 16–bit address. It can be used only the first time a .bss directive is specified. Normally, the SPC is set to 0 the first time a named section is assembled; you can use the address parameter to assign an initial value to the SPC. This parameter has no effect on the final address of the section.

Section directives for initialized sections (.text, .data, and .sect) end the current section and begin assembling into another section. Section directives for uninitialized sections (.bss, .reg, .regpair, and .usect), however, do not affect the current section. The assembler assembles the .bss, .reg, .regpair, or .usect directive and then resumes assembling code into the same section.

Example This example shows the .bss directive used to allocate space for two variables, array and dflag. The symbol array points to 100 bytes of uninitialized space (at .bss–SPC = 0). The symbol dflag points to 1 byte of uninitialized space (at .bss–SPC = 100). Note that symbols declared with the .bss directive can be referenced in the same manner as other symbols and can also be declared global.

```
 1              ;**************************************
 2              ;* Begin assembling into .text        *
 3              ;**************************************
 4 0000              .text
 5 0000  4a0b        mov   R10, R11
 6
 7              ;**************************************
 8              ;* Allocate 100 bytes in .bss         *
 9              ;**************************************
10 0000              .bss  array,100
11 0002  4b0c        mov   R11, R12 ; assembled into .text
12
13              ;**************************************
14              ;* Allocate 1 byte in .bss            *
15              ;**************************************
16 0064              .bss  dflag
17 0004 -4014005e    mov   dflag,R4 ;assembled into .text
18
19              ;**************************************
20              ;* Declare external .bss symbol       *
21              ;**************************************
22                   .global array  ; still in .text
```

Syntax

.byte *value*$_1$ [*, ... , value*$_n$]
.string *value*$_1$ [*, ... , value*$_n$]

Description

The .byte and .string directives place one or more 8–bit values into consecutive bytes of the current section. A *value* can be either :

- An expression that the assembler evaluates and treats as an 8–bit signed number, or
- A character string enclosed in double quotes. Each character in a string represents a separate value.

The assembler truncates values that are greater than 8 bits. You can use up to 100 value parameters, but the total line length cannot exceed 200 characters.

If you use a label, it points to the location at which the assembler places the first byte.

Note that when you use .byte or .string in a .struct/.endstruct sequence, .byte or .string defines a member's size; it does not initialize memory.

Example

This example shows several 8–bit values placed into consecutive bytes in memory. The label strx has the value 0h, which is the location of the first initialized byte. The label stry has the value 6h, which is the first byte initialized by the .string directive.

```
 1 0000   0a          strx    .byte   10,-1,"abc",'a'
   0001   ff
   0002   61
   0003   62
   0004   63
   0005   61
 2 0006   0a          stry    .string 10,-1,"abc",'a'
   0007   ff
   0008   61
   0009   62
   000a   63
   000b   61
```

Syntax

.copy [*"*]*filename*[*"*]
.include [*"*]*filename*[*"*]

(The quote marks surrounding the filename are optional.)

Description

The .copy and .include directives tell the assembler to read source statements from a different file. The assembler:

1) Stops assembling statements in the current source file.
2) Assembles the statements in the copied/included file.
3) Resumes assembling statements in the main source file, starting with the statement that follows the .copy or .include directive.

The *filename* is a required parameter that names a source file; the *filename* may be enclosed in double quotes. The *filename* must follow operating system conventions. You can specify a full pathname (for example, c:\430\file1.asm). If you do not specify a full pathname, the assembler searches for the file in:

1) The directory that contains the current source file.
2) Any directories named with the -i assembler option.
3) Any directories specified by the environment variable A_DIR.

The statements that are assembled from a copy file are printed in the assembly listing. The statements that are assembled from an included file are *not* printed in the assembly listing, regardless of the number of .list/.nolist directives that are assembled.

The .copy and .include directives can be nested within a file being copied or included. The assembler limits this type of nesting to eight levels; the host operating system may set additional restrictions. The assembler precedes the line numbers of copied files with a letter code to identify the level of copying. An **A** indicates the first copied file, **B** indicates a second copied file, etc.

Example 1 This example shows how the .copy directive is used to tell the assembler to read and assemble source statements from other files, then to resume assembling into the current file.

copy.asm (source file)	**byte.asm (first copy file)**	**word.asm (second copy file)**

`.space 29`
`.copy "byte.asm"`
 	
`**Back in original file`
`.string "done"`
	`**In byte.asm`
	`.byte 32,1+'A'`
	`.copy "word.asm"`
	`** Back in byte.asm`
	`.byte 67h+3q`
		`** In word.asm`
		`.word 0ABCDh, 56q`

Listing file:

```
                1 0000                          .space  29
                2                               .copy   "byte.asm"
 A              1                             ;** In byte.asm
 A              2 001d   20                     .byte 32,1+'A'
                  001e   42
 A              3                               .copy "word.asm"
 B              1                             ;** In word.asm
 B              2 0020   abcd                   .word 0ABCDh, 56q
                  0022   002e
 A              4                             ;** Back in byte.asm
 A              5 0024   6a                     .byte 67h+3q
                3
                4                             ;** Back in original file
                5
                6 0025   64                     .string "done"
                  0026   6f
                  0027   6e
                  0028   65
```

Example 2 This example shows how the .include directive is used to tell the assembler to read and assemble source statements from other files, then to resume assembling into the current file.

include.asm (source file)	**byte2.asm (first include file)**	**word2.asm (second include file)**

`.space 29`
`.include "byte2.asm"`
 	
`**Back in original file`
`.string "done"`
	`** In byte2.asm`
	`.byte 32,1+ 'A'`
	`.include "word2.asm"`
	`** Back in byte.asm`
	`.byte 67h+3q`
		`** In word2.asm`
		`.word 0ABCDh, 56q`

Listing file:

```
1 0000                          .space 29
2                               .include "byte2.asm"
3
4                               ;**Back in original file
5 0025   64                     .string "done"
  0026   6f
  0027   6e
  0028   65
```

Syntax **.data** [*address*]

Description The .data directive tells the assembler to begin assembling source code into the .data section; .data becomes the current section. The .data section is normally used to contain tables of data or preinitialized variables.

The *address* is an optional parameter that specifies a 16–bit address. It can be used only the first time a .data directive is specified. Normally, the section program counter is set to 0 the first time the .data section is assembled; you can use this parameter to assign an initial value to the .data section program counter. This parameter has no effect on the final address of the section; it simply makes the listing easier to read.

Note that the assembler assumes that .text is the default section. Therefore, at the beginning of an assembly, the assembler assembles code into the .text section unless you specify an explicit section control directive.

Example This example shows the assembly of code into the .data and .text sections.

```
 1                  ;***********************************
 2                  ;**   Reserve space in .data       **
 3                  ;***********************************
 4 0000                      .data
 5 0000                      .space 0cch
 6
 7                  ;***********************************
 8                  ;**   Assemble into .text          **
 9                  ;***********************************
10 0000                      .text
11         00       Index   .equ   0
12 0000    4304              mov    #Index, R4
13
14                  ;***********************************
15                  ;**   Assemble into .data          **
16                  ;***********************************
17 00cc             Table:  .data
18 00cc    ffff             .word  -1
19 00ce    ff               .byte  0ffh
20
21                  ;***********************************
22                  ;**   Assemble into .text          **
23                  ;***********************************
24 0002                     .text
25 0002  "901400c8          cmp Table, R4
26
27                  ;***********************************
28                  ;**   Resume Assembling into .data **
29                  ;***********************************
30 00cf                     .data
```

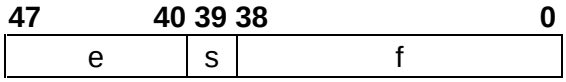
Syntax **.double** value

Description The .double directive places the floating-point representation of a double floating-point constant into six bytes in the current section. The value must be a floating-point constant. Each constant is converted to a floating-point value in the MSP430 (48-bit) format.

The 48-bit value consists of three fields:

- An 8-bit biased exponent (e)
- A 1-bit sign field (s)
- A 39-bit fraction (f)

The value is stored exponent byte first, most significant byte of fraction second, and least significant byte of fraction sixth in the following format:



Note that when you use .double in a .struct/.endstruct sequence, .double defines a member's size; it does not initialize memory. For more information about MSP430 floating-point format, refer to Appendix G.

Examples Here are some examples of the .double directive.

```
1 0000    d38459516140          .double  -1.0e25
2 0006    814000000000          .double  3
3 000c    8d40e4000000          .double  12345
```

Syntax **.emsg** *string*

.mmsg *string*

.wmsg *string*

Description Use these directives to define your own error and warning messages. Note that the assembler tracks the number of errors and warnings it encounters and prints these numbers on the last line of the listing file.

- The **.emsg** directive sends error messages to the standard output device. The .emsg directive generates errors in the same way the assembler does, incrementing the error count and preventing the assembler from producing an object file.
- The **.wmsg** directive sends warning messages to the standard output device. The .wmsg directive functions in the same way the .emsg directive does, but increments the warning count. The assembler is not prevented from producing an object file.
- The **.mmsg** directive sends assembly–time messages to the standard output device. The .mmsg directive functions in the same way the .emsg and .wmsg directives do, but does not set the error count or the warning count. The assembler is not prevented from producing an object file.

Example In this example, the message "ERROR -- MISSING PARAMETER" is sent to the standard output device.

```
  1          MSG_EX  .macro parm1
  2                  .if    $symlen(parm1) = 0
  3                  .emsg  "ERROR -- MISSING PARAMETER"
  4                  .else
  5                  MOV    parm1, A
  6                  .endif
  7                  .endm
  8
  9 0000             MSG_EX
1                    .if    $symlen(parm1) = 0
1                    .emsg  "ERROR -- MISSING PARAMETER"
1                    .else
1                    MOV    parm1, A
1                    .endif
```

These messages will show in the readout like any other error message:

```
********* USER ERROR - ERROR -- MISSING PARAMETER
********* USER WARNING - ···
********* USER MESSAGE - ···
```

Syntax **.end**

Description The .end directive is an optional directive that terminates assembly. It should be the last source statement of a program. The assembler ignores any source statements that follow an .end directive.

This directive has the same effect as an end–of–file. You can also use .end when you're debugging code and you'd like to stop assembling at a specific point in your code.

Note: Use .endm to End a Macro
Do not use the .end directive to terminate a macro; use the endm macro directive instead.

Example This example shows how the .end directive terminates assembly. If any source statements follow the .end directive, the assembler ignores them.

```
 1 0000                  Text_start:     .text
 2 0000   0a                             .byte 0ah
 3 0002   aaaa                           .word 0aaaah
 4 0004   61                             .string "aaa"
   0005   61
   0006   61
 5                                       .end
```

Syntax **.even**

Description The .even directive aligns the section program counter (SPC) on the next word boundary. This directive can be used to force the start of the next initialized data on an even address.

Example This example shows the initialization of two strings, each aligned on a word boundary. Without the .even directive the second string would start immediately after the first one (on an odd address).

```
1 0000  41          .string     "ABC"
  0001  42
  0002  43
2                   .even
3 0004  58          .string     "XYZ"
  0005  59
  0006  5a
4
```

Note: Automatic Alignment on Word Boundary

Instructions itself and data created by .word, .float and .double directives will be aligned on a word boundary automatically.

Syntax **.fclist**

.fcnolist

Description Two directives enable you to control the listing of false conditional blocks.

- The **.fclist** directive allows the listing of conditional blocks that do not produce code (false blocks). *By default, the assembler behaves as if you had used .fclist.*
- The **.fcnolist** directive inhibits the listing of false conditional blocks that do not produce code. Only code in the conditional block that actually assembles appears in the listing. The .if, .elseif, .else, and .endif directives do not appear.

Example This example shows the assembly language file and the listing file for code with and without the conditional blocks listed. This is the un-assembled file:

```
x  .set 1          ;True
y  .set 0          ;False

   .fclist

   .if x
   MOV #5, R4
   .else
   MOV #9, R4
   .endif

   .fcnolist

   .if x
   MOV #5, R4
   .else
   MOV #9, R4
   .endif
```

This is the listing file:

```
 1        01             x   .set 1   ;True
 2        00             y   .set 0   ;False
 3
 4                            .fclist
 5
 6                            .if x
 7 0000   40340005            MOV #5, R4
 8                            .else
 9                            MOV #9, R4
10                            .endif
11
12                            .fcnolist
13
15 0004   40340005            MOV #5, R4
```

Syntax **.field** *value* [*, size in bits*]

Description The .field directive initializes multiple–bit fields within a single word of memory. This directive has two operands:

- The *value* is a required parameter; it is an expression that is evaluated and placed in the field. The value must be absolute.

- The *size* is an optional parameter; it specifies a number from 1 to 16, which is the number of bits the field consists of. If you do not specify a size, the assembler assumes that the size is 16 bits. If you specify a value that cannot fit in *size* bits, the assembler truncates the value and issues a warning message. For example, .field 3,1 causes the assembler to truncate the value *3* to 1; the assembler also prints the message:

```
***warning - value truncated.
```

Successive .field directives pack values into the specified number of bits in the current word. Fields are packed starting at the least significant part of the word, moving toward the most significant part as more fields are added. If the assembler encounters a field size that does not fit into the current word, it writes out the word, increments the SPC, and begins packing fields into the next word.

You can use the .align directive to force the next .field directive to begin packing into a new byte.

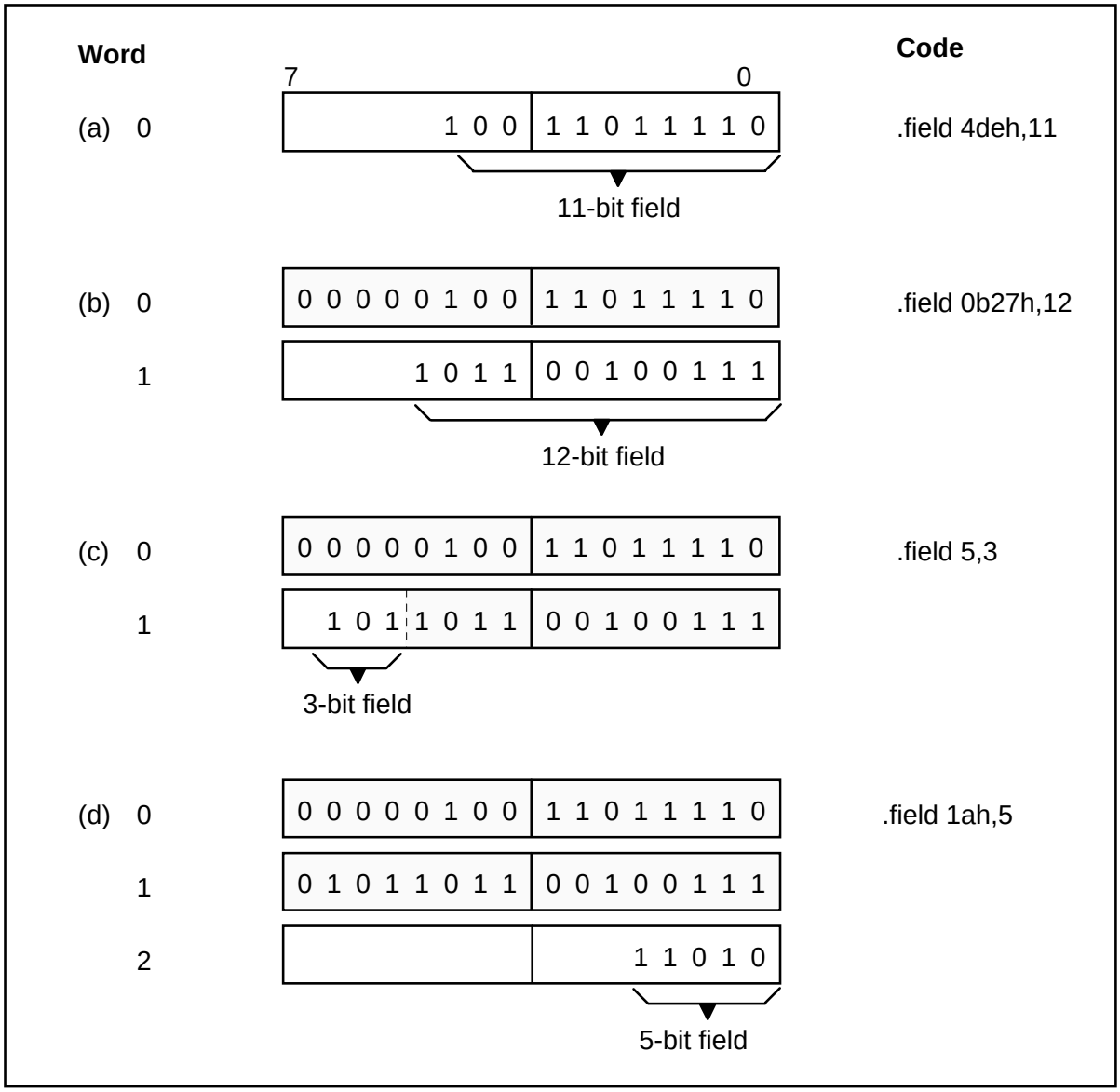
If you use a label, it points to the word that contains the field.

Note also that when you use .field in a .struct/.endstruct sequence, .field defines a member's size; it does not initialize memory.

Example This example shows how fields are packed into a word. Note that the SPC does not change until a word is filled and the next word is begun.

```
 1                      ********************************
 2                      *   Initialize a 11-bit field  *
 3                      ********************************
 4 0000   04de                  .field  4deh,11
 5
 6                      ********************************
 7                      *   Initialize a 12-bit field  *
 8                      *   in a new word              *
 9                      ********************************
10 0000   0b27                  .field  0b27h,12
11
12                      ********************************
13                      *   Initialize a 3-bit field   *
14                      *   in the same byte           *
15                      ********************************
16 0000   5b27                  .field  5,3
17
18                      ********************************
19                      *   Initialize a 5-bit field   *
20                      *   in the next byte           *
21                      ********************************
22 0000   1a                    .field 1ah,5
```

The example shows how the directives affect memory.



Example 4.7: The .field Directive

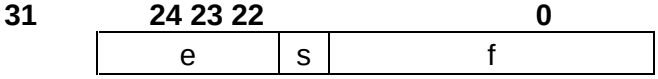
Syntax **.float** *value*

Description The .float directive places the floating–point representation of a single floating–point constant into four bytes in the current section. The *value* must be a floating–point constant. Each constant is converted to a floating–point value in MSP430 (32–bit) format.

The 32–bit value consists of three fields:

- An 8–bit biased exponent (*e*)
- A 1–bit sign field (*s*)
- A 23–bit fraction (*f*)

The value is stored exponent byte first, most significant byte of fraction second, and least significant byte of fraction fourth in the following format:



Note that when you use .float in a .struct/.endstruct sequence, .float defines a member's size; it does not initialize memory. For more information about MSP430 floating–point format, refer to Appendix G.

Example Here are some examples of the .float directive.

```
1 0000   d3845951              .float   -1.0e25
2 0004   81400000              .float   3
3 0008   8d40e400              .float   12345
```

Syntax

.global *symbol*$_1$ [*, ... , symbol*$_n$]

.def *symbol*$_1$ [*, ... , symbol*$_n$]

.ref *symbol*$_1$ [*, ... , symbol*$_n$]

Description

The .global, .def, and .ref directives identify global symbols, which are defined externally or can be referenced externally.

- The **.def** directive identifies a symbol that is defined in the current module and can be accessed by other files. The assembler places this symbol in the symbol table.
- The **.ref** directive identifies a symbol that is used in the current module but defined in another module. The linker resolves this symbol's definition at link time.
- The **.global** directive acts as a .ref or a .def, as needed.

A global symbol is *defined* in the same manner as any other symbol; that is, it appears as a label or is defined by the .set, .equ, .bss or .usect directive. As with all symbols, if a global symbol is defined more than once, the linker issues a multiple–definition error. Note that .ref always creates an entry for a symbol, whether the module uses the symbol or not; .global, however, create a symbol table entry only if the module actually uses the symbol.

A symbol may be declared global for two reasons:

1) If the symbol is *not defined in the current module* (including macro, copy, and include files), the .global or .ref directive tells the assembler that the symbol is defined in an external module. This prevents the assembler from issuing an unresolved reference error. At link time, the linker looks for the symbol's definition in other modules.
2) If the symbol *is defined in the current module*, the .global, .globreg, or .def directive declares that the symbol and its definition can be used externally by other modules. These types of references are resolved at link time.

Example

This example shows four files:

- file1.lst and file3.lst are equivalent. Both files define the symbol Init and make it available to other modules; both files use the external symbols x, y, and z. file1.lst uses the .global directive to identify the global symbols; file3.lst uses .ref and .def to identify the symbols.
- file2.lst and file4.lst are equivalent. Both files define the symbols x, y, and z and make them available to other modules; both files use the external symbol Init. file2.lst uses the .global directive to identify the nonregister global symbols; file4.lst uses .ref and .def to identify the nonregister symbols.

file1.lst

```
 1                    ; Global symbol defined in this file
 2                              .global init
 3
 4                    ; Global symbols defined in file2.lst
 5                              .global x, y, z
 6
 7 0000   5034002c    init:     add #44, R4
 8 0004  !0000                   .word x
 9                    ;  .
10                    ;  .
11                    ;  .
12                              .end
```

file2.lst

```
 1                    ; Global symbol defined in this file
 2                              .global x, y, z
 3
 4                    ; Global symbols defined in file1.lst
 5                              .global init
 6
 7        01          x        .equ    1
 8        02          y        .equ    2
 9        03          z        .equ    3
10 0000  !0000                  .word   init
11                    ;  .
12                    ;  .
13                    ;  .
14                              .end
```

file3.lst

```
 1                      ; Global symbol defined in this file
 2                               .def init
 3
 4                      ; Global symbols defined in file4.lst
 5                               .ref x, y, z
 6
 7 0000   5034002c      init:   add #44, R4
 8 0004  !0000                   .word x
 9                      ;  .
10                      ;  .
11                      ;  .
12                              .end
```

file4.lst

```
 1                      ; Global symbol defined in this file
 2                               .def x, y, z
 3
 4                      ; Global symbols defined in file3.lst
 5                               .ref init
 6
 7        01            x       .equ    1
 8        02            y       .equ    2
 9        03            z       .equ    3
10 0000  !0000                   .word   init
11                      ;  .
12                      ;  .
13                      ;  .
14                              .end
```

Syntax

.if *well–defined expression*
code to assemble when the expression is true

.elseif *well–defined expression*
code block to execute when the expression is true

.else
code to assemble when the expression is false

.endif
terminate condition block

Description

Four directives provide conditional assembly:

- The **.if** directive marks the beginning of a conditional block. The *expression* is a required parameter.
  - If the expression evaluates to *true* (nonzero), the assembler assembles the code that follows it (up to an .elseif, an .else, or an .endif).
  - If the expression evaluates to *false* (0), the assembler assembles code that follows an .elseif (if present), an else (if present), or an .endif (if no .elseif or .else is present).
- The **.elseif** directive identifies a block of code to be assembled when the .if expression is false (0) and the .elseif expression is true (nonzero). When the .elseif expression is false, the assembler continues to the next .elseif (if present), .else (if present), or an .endif. The .elseif directive is optional in the conditional block, and more than one .elseif can be used. If an expression is false and there is no .elseif statement, the assembler continues with the code that follows an .else (if present) or an .endif.
- The **.else** directive identifies a block of code that the assembler assembles when the .if expression is false (0). This directive is optional in the conditional block; if an expression is false and there is no .else statement, the assembler continues with the code that follows the .endif.
- The **.endif** directive terminates a conditional block.

The .elseif and .else directives can be used in the same conditional assembly block, and the .elseif directive can be used more than once within a conditional assembly block.

Example Here are some examples of conditional assembly:

```
 1       01    sym1  .set     1
 2       02    sym2  .set     2
 3       03    sym3  .set     3
 4       04    sym4  .set     4
 5             If_4: .if   sym4=sym2*sym2
 6 0000  04          .byte sym4        ;  Equal values
 7                   .else
 8                   .byte sym2 * sym2 ; Unequal values
 9                   .endif
10             If_5: .if   sym1<=10
11 0001  0A          .byte 10          ; Less than/equal
12                   .else
13                   .byte sym1        ; Greater than
14                   .endif
15             If_6: .if   sym3*sym2!=sym4+sym2
16                   .byte sym3*sym2   ;Unequal values
17                   .else
18 0002  06          .byte sym4+sym2   ;Equal values
19                   .endif
20             If_7  .if   sym1=2
21                   .byte sym1
22                   .elseif sym2+sym3=5
23 0003  05          .byte sym2+sym3
24                   .endif
```

Syntax **.label** *symbol*

Description The .label directive defines a special symbol that refers to the loadtime address rather than the runtime address within the current section. Most sections created by the assembler have relocatable addresses. The assembler assembles each section as if it started at zero, and the linker relocates it to the address at which it loaded and ran.

For some applications, it is desirable to have a section load at one address and run at a **different** address. For example, you may wish to load a block of performance–critical code into slower off–chip memory to save space, and then move the code to high–speed on–chip memory to run it.

Such a section is assigned two addresses at link time: a load address and a separate run address. All labels defined in the section are relocated to refer to the runtime address so that references to the section (such as branches) are correct when the code runs.

The .label directive creates a special “label" that refers to the *loadtime* address. This is useful primarily so that the code that relocates the section knows where the section was loaded. For example:

```
;-----------------------------------------
; .label Example
;-----------------------------------------
      .sect ".examp"
      .label   examp_load  ; load address of section
start:                     ; run address of section
      <code>
finish:                    ; run address of section end
      .label   examp_end   ; load address of section end
```

Syntax **.length** *page length*
.width *page width*

Description The **.length** directive sets the page length of the output listing file. It affects the current page and following pages; you can reset the page length with another .length directive.

- Default length: 60 lines
- Minimum length: 1 line
- Maximum length: 32,767 lines

The **.width** directive sets the page width of the output listing file. It affects the next line assembled and following lines; you can reset the page width with another .width directive.

- Default width: 132 characters
- Minimum width: 80 characters
- Maximum width: 200 characters

Note that the width refers to a full line in a listing file; the line counter value, SPC value, and object code are counted as part of the width of a line. Comments and other portions of a source statement that extend beyond the page width are truncated in the listing.

The assembler does not list the .width and .length directives.

Example The following example shows how to change the page length and width.

```
*****************************************
**        Page length = 65 lines       **
**        Page width  = 85 characters  **
*****************************************
          .length    65
          .width     85

*****************************************
**        Page length = 55 lines       **
**        Page width  = 100 characters **
*****************************************
          .length    55
          .width     100
```

Syntax **.list**
.nolist

Description The .nolist directive suppresses the source listing output until a .list directive is encountered. The .list directive tells the assembler to resume printing the source listing after it has been stopped by a .nolist directive. *By default, the assembler acts as if a .list directive had been specified.* The .nolist directive can be used to reduce assembly time and the size of the source listing; it can be used in macro definitions to inhibit the listing of the macro expansion.

The assembler does not print the .list or .nolist directives or the source statements that appear after a .nolist directive; however, it continues to increment the line counter. You can nest the .list/.nolist directives; each .nolist needs a matching .list to restore the listing. At the beginning of an assembly, the assembler acts as if it has assembled a .list directive.

Note: Creating a Listing File (-l option)

If you don't request a listing file when you invoke the assembler, the assembler ignores the .list directive.

Example This example shows how to use the .copy directive to insert source statements from another file. The first time this directive is encountered, the assembler lists the copied source lines in the listing file. The second time this directive is encountered, the assembler does not list the copied source lines, because a .nolist directive was assembled. Note that the .nolist, the second .copy, and .list directives do not appear in the listing file; note also that the line counter is incremented even when source statements are not listed.

Source file:

```
        .copy    "copy2.asm"
* Back in original file
        .nolist
        .copy    "copy2.asm"
        .list
* Back in original file
        .string  "Done"
```

Listing file:

```
       1                                        .copy "copy2.asm"
A      1                                * In copy2.asm (copy file)
A      2 0000   0020                            .word 32, 1+'A'
         0002   0042
       2                                * Back in original file
       6                                * Back in original file
       7 0008   44                               .string "Done"
         0009   6f
         000a   6e
         000b   65
```

Syntax

.loop [*well–defined expression*]
code block to repeatedly assemble

.break [*well–defined expression*]
continue to assemble repeatedly when the .break expression is false (zero); go to code immediately following .endloop if expression is true (nonzero)

.endloop
code block to execute when the .break directive is true (nonzero) or when the .break expression is omitted and the loop count equals the expression

Description

Three directives enable you to repeatedly assemble a block of code:

- The **.loop** directive begins a repeatable block of code. The optional expression evaluates to the loop count. If there is no expression, the loop count defaults to 1024, unless the assembler encounters a .break directive.
- The **.break** directive is optional, along with its expression. When the expression is false (0), the loop continues. When the expression is true (nonzero) or omitted, the assembler breaks the loop and assembles the code after the .endloop directive.
- The **.endloop** directive terminates a repeatable block of code.

Example This example illustrates how these directives can be used with the .eval directive.

```
  1                                    .eval   0, x
  2                          coef      .loop
  3                                    .word   x*100
  4                                    .eval   x+1,x
  5                                    .break  x = 7
  6                                    .endloop
1          0000    0000                .word   0*100
1                                      .eval   0+1,x
1                                      .break  1 = 7
1          0002    0064                .word   1*100
1                                      .eval   1+1,x
1                                      .break  2 = 7
1          0004    00c8                .word   2*100
1                                      .eval   2+1,x
1                                      .break  3 = 7
1          0006    012c                .word   3*100
1                                      .eval   3+1,x
1                                      .break  4 = 7
1          0008    0190                .word   4*100
1                                      .eval   4+1,x
1                                      .break  5 = 7
1          000a    01f4                .word   5*100
1                                      .eval   5+1,x
1                                      .break  6 = 7
1          000c    0258                .word   6*100
1                                      .eval   6+1,x
1                                      .break  7 = 7
```

Syntax **.mlib** ['']*filename*['']

(The quote marks surrounding the filename are optional.)

Description The .mlib directive provides the assembler with the name of a macro library. A macro library is a collection of files that contain macro definitions. These files are bound into a single file (called a library or archive) by the archiver. Each member of a macro library may contain one macro definition that corresponds to the name of the file. Note that:

- Macro library members must be **source** files (not object files).
- The filename of a macro library member must be the same as the macro name, and its extension must be .asm.

The *filename* must follow host operating system conventions; it may be enclosed in double quotes. You can specify a full pathname (for example, c:\430\macs.lib). If you do not specify a full pathname, the assembler searches for the file in:

1) The directory that contains the current source file.
2) Any directories named with the -i assembler option.
3) Any directories specified by the environment variable A_DIR .

When the assembler encounters an .mlib directive, it opens the library and creates a table of the library's contents. The assembler enters the names of the individual library members into the opcode table as library entries; this redefines any existing opcodes or macros that have the same name. If one of these macros is called, the assembler extracts the entry from the library and loads it into the macro table. The assembler expands the library entry in the same manner as other macros, but it does not place the source code into the listing. Only macros that are actually called from the library are extracted, and they are extracted only once.

Example This example shows how to create a macro library that defines two macros, inc1 and dec1. The file inc1.asm contains the definition of inc1, and dec1.asm contains the definition of dec1.

inc1.asm

```
* Macro for incrementing
inc1  .MACRO nam
      mov    nam, R4
      inc    R4
      mov    R4, nam
      .ENDM
```

dec1.asm

```
* Macro for decrementing
dec1  .MACRO nam
      mov    nam, R4
      dec    R4
      mov    R4, nam
      .ENDM
```

```
ar430 -a mac inc1.asm dec1.asm
```

Now you can use the .mlib directive to reference the macro library and define the inc1 and dec1 macros:

```
        1 0000                .bss    var1,1
        2 0001                .bss    var2,1
        3
        4                     .mlib   "mac.lib"
        5
        6 0000                inc1    var1    ; macro call
1         0000  -40140000     mov    var1, R4
1         0004   5314         inc    R4
1         0006  -44800000     mov    R4, var1
        7 000a                dec1    var2    ; macro call
1         000a  -4014fff5     mov    var2, R4
1         000e   8314         dec    R4
1         0010  -4480ffef     mov    R4, var2
```

Syntax **.mlist**
.mnolist

Description Two directives provide you with the ability to control the listing of macro and repeatable block expansions in the listing file:

- The **.mlist** directive allows macro and .loop/.endloop block expansions in the listing file.
- The **.mnolist** directive inhibits macro and .loop/.endloop block expansions in the listing file.

By default, all code encountered in macros and .loop/.endloop blocks is listed.

Example This example shows how to define a macro named str_3. The first time the macro is called, the macro expansion is listed (by default). The second time the macro is called, the macro expansion is not listed, because a .mnolist directive was assembled. The third time the macro is called, the macro expansion is again listed because a .mlist directive was assembled.

```
 1              str_3   .MACRO  pm1, pm2, pm3
 2                      .string ":pm1:", ":pm2:", ":pm3:"
 3                      .ENDM
 4
 5 0000                 str_3   "as","I","am"
1  0000   61            .string "as", "I", "am"
   0001   73
   0002   49
   0003   61
   0004   6d
 6                      .mnolist
 7 0005                 str_3   "as","I","am"
 8                      .mlist
 9 000a                 str_3   "as","I","am"
1  000a   61            .string "as", "I", "am"
   000b   73
   000c   49
   000d   61
   000e   6d
```

Syntax **.newblock**

Description The .newblock directive undefines any local labels currently defined. A local label, by nature, is temporary; the .newblock directive resets local labels and terminates their scope.

A local label is a label in the form $*n*, where *n* is a single decimal digit. A local label, like other labels, points to an instruction byte. Unlike other labels, local labels cannot be used in expressions; they can be used only as the operand in 10-bit jump instructions. Local labels are not included in the symbol table.

After a local label has been defined and (perhaps) used, you should use the .newblock directive to reset it. Note that the .text, .data, and .sect directives also reset local labels and that local labels that are defined within an include file are not valid outside of the include file.

Example This example shows how the local label $1 is declared, reset, and then declared again.

```
 1 0000   4c0d       Label1: mov     R12, R13
 2 0002   2000               jnz     $1
 3 0004   433d               mov     #-1, r13
 4 0006   9d04       $1      cmp     R13, R4
 5                           .newblock       ;undefine $1
 6 0008   2000               jne     $1
 7 000a   531d               inc     R13
 8 000c   5d0e       $1      add     R13, R14
```

Syntax **.option** *option list*

Description The .option directive selects several options for the assembler output listing. The *option list* is a list of options separated by commas; each option selects a listing feature. Valid options include:

A Turns on all listing (overrides all other directives and options).

B Limits the listing of .byte directives to one line.

F Resets the B, M, W, and T options.

M Turns off macro expansions in the listing.

T Limits the listing of .string directives to one line.

W Limits the listing of .word directives to one line.

X Produces a symbol cross–reference listing.

Options **are not** case–sensitive.

Example This example shows how to limit the listings of the .byte, .word, and .string directives to one line each.

```
 1                      ;****************************************
 2                      ;*  Limit the listing of .byte, .word,  *
 3                      ;*  .string directives to 1 line each   *
 4                      ;****************************************
 5                               .option B, W, T
 6 0000   bd                     .byte   -'C', 0B0h, 5
 7 0004   15aa                   .word   5546, 78h
 8 0008   59                     .string "YES"
 9                      ;****************************************
10                      ;* Reset the listing options            *
11                      ;****************************************
12                               .option F
13 000b   bd                     .byte   -'C', 0B0h, 5
   000c   b0
   000d   05
14 000e   15aa                   .word   5546, 78h
   0010   0078
15 0012   59                     .string "YES"
   0013   45
   0014   53
16                      ;****************************************
17                      ;*  Use The A option to ignore all      *
18                      ;*  other options and directives        *
19                      ;****************************************
20                               .option A
21                               .nolist
22                               .option B, W, T
23 0015   bd                     .byte   -'C', 0B0h, 5
   0016   b0
   0017   05
24 0018   15aa                   .word   5546, 78h
   001a   0078
25 001c   59                     .string "YES"
   001d   45
   001e   53
26                               .list
```

Syntax **.page**

Description The .page directive produces a page eject in the listing file. The .page directive is not printed in the source listing, but the line counter is incremented. Using the .page directive to divide the source listing into logical divisions improves program readability.

Example This example shows how the .page directive causes the assembler to begin a new page of the source listing.

Source file:

```
        .title   "**** Page Directive Example ****"
;         .
;         .
;         .
        .page
```

Listing file:

```
MSP TSS430 Macro Assembler Prototype Version 1.0 [Mar 22] Wed Aug 18 08:34:18 1993
 Copyright (c) 1993    Texas Instruments Incorporated

**** Page Directive Example ****                                    PAGE    1

       2                        ;         .
       3                        ;         .
       4                        ;         .
MSP TSS430 Macro Assembler Prototype Version 1.0 [Mar 22] Wed Aug 18 08:34:18 1993
 Copyright (c) 1993    Texas Instruments Incorporated

**** Page Directive Example ****                                    PAGE    2
```

Syntax **.sect "***section name***"**[,*address*]

Description The .sect directive defines a named section that can be used like the default .text and .data sections. The .sect directive begins assembling source code into the named section.

- The *section name* identifies a section that the assembler assembles code into. The *name* is significant to 8 characters and must be enclosed in double quotes.
- The *address* is an optional parameter that specifies a 16–bit address. It can be used only the first time a .sect directive is specified for a particular section. Normally, the SPC is set to 0 the first time a named section is assembled; you can use the address parameter to assign an initial value to the SPC. This parameter has no effect on the final address of the section; it simply makes the listing easier to read.

Example This example shows how two special–purpose sections, Sym_Defs and Vars, are defined, and how code assembles into them.

```
 1                      ;************************************
 2                      ;*  Begin assembling into .text     *
 3                      ;************************************
 4 0000                          .text
 5 0000   4b0c                   MOV R11, R12
 6 0002   4d0e                   MOV R13, R14
 7
 8                      ;************************************
 9                      ;*  Begin assembling into Sym_Defs  *
10                      ;************************************
11 0000                          .sect   "Sym_Defs"
12 0000   00aa     X             .word   0aah
13 0002   50340005               ADD     #5, R4
14
15                      ;************************************
16                      ;*  Begin assembling into Vars      *
17                      ;************************************
18 4000                          .sect "Vars", 4000h
19        10       Word_Len      .set  16
20        08       Byte_Len      .set  16 / 2
21
22                      ;************************************
23                      ;*  Resume assembling into .text    *
24                      ;************************************
25 0004                          .text
26 0004   5034002a               ADD #42, R4
27 0008   03                     .byte 3,4
          0009   04
28                      ;************************************
29                      ;*  Resume assembling into Vars     *
30                      ;************************************
31 4000                          .sect "Vars"
32 4000   01                     .field 1,2
33 4000   09                     .field 2,2
```

Syntax

symbol **.set** *value*
symbol **.equ** *value*

Description

The .set and .equ directives equate a constant value to a symbol. The symbol can then be used in place of a value in assembly source. This allows you to equate meaningful names with constants and other values. The .set and .equ directives are identical and can be used interchangeably.

- The *symbol* must appear in the label field.
- The *value* must be a well–defined expression; that is, all symbols in the expression must be previously defined in the current source module.

Undefined external symbols and symbols that are defined later in the module cannot be used in the expression. If the expression is relocatable, the symbol to which it is assigned is also relocatable.

The value of the expression appears in the object field of the listing. This value is not part of the actual object code and is not written to the output file.

Example This example shows how symbols can be assigned with .set.

```
 1                      ;************************************
 2                      ;* Equate symbol FP to register R11  *
 3                      ;*  and use it instead of R11        *
 4                      ;************************************
 5          0b          FP      .set    R11
 6 0000     4b24                MOV     @FP, R4
 7
 8                      ;************************************
 9                      ;* Set symbol count to an integer    *
10                      ;* expression and use it as an       *
11                      ;* immediate operand                 *
12                      ;************************************
13          35          count   .equ    100/2+3
14 0002    40340035             MOV     #count, R4
15
16                      ;************************************
17                      ;* Set symbol symtab to relocatable  *
18                      ;* expression                        *
19                      ;************************************
20 0006     000a        label   .word   10
21         '07          symtab  .set    label+1
22 0008    '0007                .word   symtab
23
24                      ;************************************
25                      ;* Set symbol nsyms to another       *
26                      ;* symbol (count) and use it instead *
27                      ;* of count                          *
28                      ;************************************
29          35          nsyms   .equ    count
30 000a    40340035              MOV     #nsyms, R4
```

Syntax **.space** *size in bytes*

Description The .space directive reserves ***size*** number of bytes in the current section and fills them with 0s. The section program counter is incremented to point to the byte following the reserved space.

When you use a label with the .space directive, it points to the *first* byte reserved.

Example This example shows how the .space directive reserves memory.

```
 1                    ;***********************************
 2                    ;*  Begin assembling into .text     *
 3                    ;***********************************
 4 0000                         .text
 5
 6                    ;***********************************
 7                    ;*  Reserve 15 bytes in .text       *
 8                    ;***********************************
 9 0000                         .space 0fh
10 0010   0100                  .word 100h, 200h
   0012   0200
11
12                    ;***********************************
13                    ;*  Begin assembling into .data     *
14                    ;***********************************
15 0000                          .data
16 0000   2e                     .string ".data"
   0001   64
   0002   61
   0003   74
   0004   61
17
18                    ;***********************************
19                    ;*  Reserve 100 bytes in .data;     *
20                    ;*  Res_1 points to the first byte  *
21                    ;***********************************
22 0005               Res_1      .space 100
23 006a   000f                   .word  15
24 006c  "0005                   .word  Res_1
```

Syntax **.sslist**

.ssnolist

Description Two directives enable you to control substitution symbol expansion in the listing file:

- The **.sslist** directive allows substitution symbol expansion in the listing file. The expanded line appears below the actual source line.
- The **.ssnolist** directive inhibits substitution symbol expansion in the listing file.

By default, all substitution symbol expansion in the listing file is inhibited. The lines with the pound (#) character denote expanded substitution symbols.

Example This example shows code that by default (.ssnolist directive) inhibits the listing of substitution symbol expansion, and it shows the .sslist directive assembled, which tells the assembler to list substitution symbol code expansion.

```
                1 0000                          .bss    x,1
                2 0001                          .bss    y,1
                3
                4                       ADD2    .macro  pm1, pm2
                5                               mov     pm1,R4
                6                               mov     R4, R5
                7                               mov     pm2, R4
                8                               add     R5, R4
                9                               mov     R4, pm2
               10                               .endm
               11
               12                               .asg    R11, FP
               13                               .asg    R13, SSP
               14
               15 0000   4b0d                   mov     FP, SSP
               16 0002                          add2    x, y
1                 0002  -40140000               mov     x,R4
1                 0006   4405                   mov     R4, R5
1                 0008  -4014fff7               mov     y, R4
1                 000c   5504                   add     R5, R4
1                 000e  -4480fff1               mov     R4, y
               17
               18                               .sslist
               19
               20 0012   4b0d                   mov     FP, SSP
#                                               mov     R11, R13
               21 0014                          add2    x, y
1                 0014  -40140000               mov     pm1,R4
#                                               mov     x,R4
1                 0018   4405                   mov     R4, R5
1                 001a  -4014ffe5               mov     pm2, R4
#                                               mov     y, R4
1                 001e   5504                   add     R5, R4
1                 0020  -4480ffdf               mov     R4, pm2
#                                               mov     R4, y
```

Syntax

[$stag_1$]	**.struct**		[$expr_1$]
[mem_0]	*element*		[$expr_2$]
[mem_1]	*element*		[$expr_2$]
.	.		.
.	.		.
[mem_n]	**.tag**	$stag_2$	[,$expr_2$]
.	.		.
[mem_N]	*element*		[$expr_2$]
[*size*]	**.endstruct**		
label	**.tag**		$stag_1$

Description

The .struct directive assigns symbolic offsets to the elements of a data structure definition. This enables you to group similar data elements together and then let the assembler do the element offset calculation. This is similar to a C structure or a Pascal record. The .struct directive does not allocate any memory; it merely creates a symbolic template that can be used repeatedly.

The .tag directive gives structure characteristics to a label, simplifying the symbolic representation and providing the ability to define structures that contain other structures. .tag does not allocate memory. The structure tag (stag) of a .tag directive must have been previously defined.

[$stag_1$] Is the structure's tag. Its value is associated with the beginning of the structure. If no stag is present, this tells the assembler to put the structure members in the global symbol table with their value being their absolute offset from the top of the structure.

[$expr_1$] Is an expression indicating the beginning offset of the structure. Structures default to start at 0.

[mem_n] Is a label for a member of the structure. This label is absolute and equates to the present offset from the beginning of the structure.

element Is one of the following descriptors: .string, .byte, .word, .float, .double, .tag, and .field. All of these, except .tag, are typical directives that initialize memory. Following a .struct directive, these directives describe the structure element's size. They **do not** allocate memory. A .tag directive is a special case because a stag must be specified (such as $stag_2$ in the definition).

[$expr_2$] Is an expression for the number of elements described. This value defaults to 1.

[*size*] Is a label for the total size of the structure.

Note: The Types of Directives That Can Appear in a .struct/.endstruct Sequence

The only directives that can appear in a .struct/.endstruct sequence are element descriptors, conditional assembly directives, and the .align and .even directives, which align the member offsets on byte resp. word boundaries. Note that empty structures are illegal.

Examples

```
 1      0000    real_rec  .struct         ;stag
 2      0000    nom       .byte           ;member1  = 0
 3      0001    den       .byte           ;member2  = 1
 4      0002    real_len  .endstruct      ;real_len = 2
 5
 6 0000 -4014ffff         mov   real+real_rec.den,R4     ;access
 7
 8 0000                   .bss  real,real_len            ;allocate
 9

10      0000    cplx_rec  .struct         ;stag
11      0000    reali     .tag  real_rec  ;member1  = 0
12      0002    imagi     .tag  real_rec  ;member2  = 2
13      0004    cplx_len  .endstruct      ;cplx_len = 4
14
15              complex   .tag  cplx_rec
16 0002                   .bss  complex,cplx_len         ;allocate
17
18 0004 -4014fffe         mov   complex.imagi.nom,R4
19 0008 -9014fff9         cmp   complex.reali.den,R4
20

21      0000              .struct         ;no stag puts members
22                                        ;into global symbol table
23      0000    x         .byte           ;create 3dim templates
24      0001    y         .byte
25      0002    z         .byte
26                        .endstruct
27

28      0000    bit_rec   .struct         ;stag
29      0000    stream    .string 64
30      0040    bit7      .field  7
31      0040    bit2      .field  2
32      0041    bit5      .field  5
33      0042    x_int     .byte
34      0043    bit_len   .endstruct
35
36              bits      .tag  bit_rec
37 0006                   .bss  bits,bit_len
38
39 000c -40140038         mov   bits.bit7,R4 ;load field
40 0010 f034007f          and   #7fh,R4      ;mask off garbage
```

Syntax **.text** [*address*]

Description The .text directive tells the assembler to begin assembling into the .text section, which usually contains executable code. The section program counter is set to 0 if nothing has yet been assembled into the .text section. If code has already been assembled into the .text section, the section program counter is restored to its previous value in the section.

The *address* is an optional parameter that specifies a 16–bit address. It can be used only the first time a .text directive is specified. Normally, the section program counter is set to 0 the first time the .text section is assembled; you can use this parameter to assign an initial value to the .text section program counter. This parameter has no effect on the final address of the section; it simply makes the listing easier to read.

Note that the assembler assumes that .text is the default section. Therefore, at the beginning of an assembly, the assembler assembles code into the .text section unless you specify one of the other sections directives (.data or .sect).

Example This example shows code assembled into the .text and .data sections. The .data section contains integer constants, and the .text section contains character strings.

```
 1                      ;**************************************
 2                      ;* Begin assembling into .data        *
 3                      ;**************************************
 4 0000                         .data
 5 0000   05                    .byte 5,6
   0001   06
 6
 7                      ;**************************************
 8                      ;* Begin assembling into .text        *
 9                      ;**************************************
10 7000                         .text 7000h
11 7000   01                    .byte 1
12 7001   02                    .byte 2, 3
   7002   03
13
14                      ;**************************************
15                      ;* Resume assembling into .data       *
16                      ;**************************************
17 0002                         .data
18 0002   07                    .byte 7,8
   0003   08
19
20                      ;**************************************
21                      ;* Resume assembling into .text       *
22                      ;**************************************
23 7003                         .text
24 7003   04                    .byte 4
```

Syntax **.title "***string***"**

Description The .title directive supplies a title that is printed in the heading on each listing page. The source statement itself is not printed, but the line counter is incremented. The *string* is a quote–enclosed title of up to 65 characters. If you supply more than 65 characters, the assembler truncates the string and issues a warning.

The assembler prints the title on the page that follows the directive and on subsequent pages until another .title directive is processed. If you want a title on the first page of a listing, the first source statement must contain a .title directive.

Example This example shows how to print one title on the first page and a different title on succeeding pages.

Source file:

```
        .title   "** Integer Routines **"
;         .
;         .
;         .
        .title   "** Floating Point Routines **"
        .page
```

Listing file:

```
MSP TSS430 Macro Assembler Prototype Version 1.0 [Mar 22] Wed Aug 18 11:01:02 1993
 Copyright (c) 1993    Texas Instruments Incorporated

** Integer Routines **                                          PAGE    1

       2                        ;         .
       3                        ;         .
       4                        ;         .
MSP TSS430 Macro Assembler Prototype Version 1.0 [Mar 22] Wed Aug 18 11:01:02 1993
 Copyright (c) 1993    Texas Instruments Incorporated

** Floating Point Routines **                                   PAGE    2
```

Syntax *symbol* **.usect "***section name***"***, size in bytes*

Description The .usect directive reserves space for variables in an uninitialized, named section. This directive is similar to the .bss directive; both simply reserve space for data and have no contents. .usect defines additional sections, however, that can be placed anywhere in memory, independently of the .bss section.

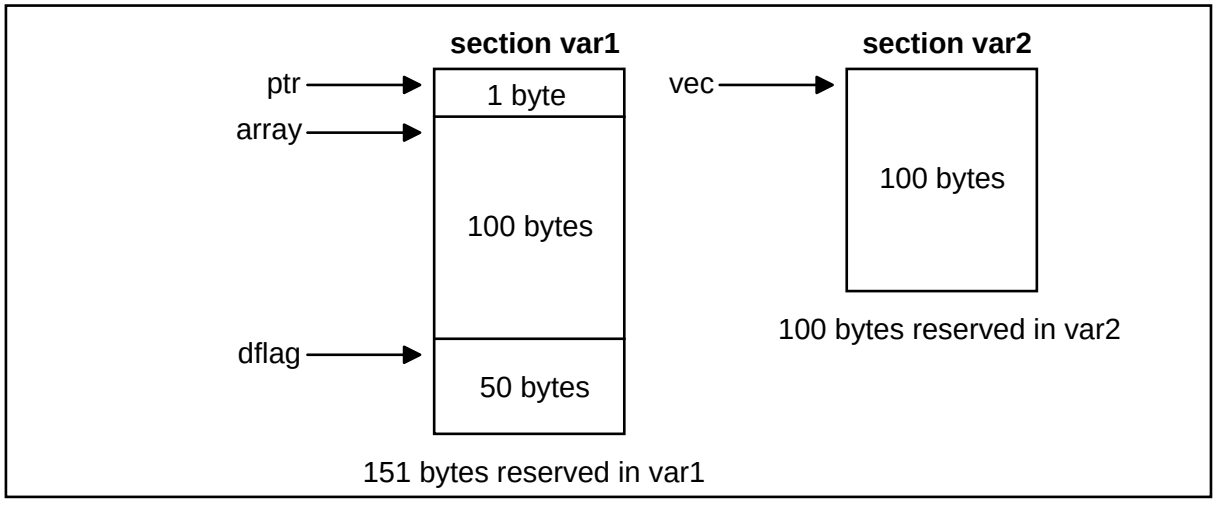
- The *symbol* points to the first location reserved by this invocation of the .usect directive. The *symbol* corresponds to the name of the variable that you're reserving space for.
- The *section name* must be enclosed in double quotes; only the first 8 characters are significant. This parameter names the uninitialized section.
- The *size* is an expression that defines the number of bytes that are reserved in section *name.*
- The *address* is an optional parameter that specifies a 16–bit address. It can be used only the first time a .usect directive is specified for a particular section. Normally, the SPC is set to 0 the first time a named section is assembled; you can use the address parameter to assign an initial value to the SPC. This parameter has no effect on the final address of the section.

Other sections directives (.text, .data, and .sect) end the current section and tell the assembler to begin assembling into another section. The .usect, .bss, .regpair, and .reg directives, however, do not affect the current section. The assembler assembles the .usect, .bss, .regpair, and .reg directives and then resumes assembling into the current section.

You can repeat the .usect directive to define more than one variable in the specified section. Variables that can be located contiguously in memory can be defined in the same section by using multiple .usect directives with the same section name.

Example This example shows how to use the .usect directive to define two uninitialized, named sections, var1 and var2. The symbol ptr points to the first byte reserved in the var1 section. The symbol array points to the first byte in a block of 100 bytes reserved in var1, and dflag points to the first byte in a block of 50 bytes in var1. The symbol vec points to the first byte reserved in the var2 section.

```
 1                      ;***********************************
 2                      ;*  Assemble into .text            *
 3                      ;***********************************
 4 0000                         .text
 5 0000   40340003              mov #03h, R4
 6
 7                      ;***********************************
 8                      ;*  Reserve 1 byte in var1         *
 9                      ;***********************************
10 0000                 ptr     .usect  "var1",1
11
12                      ;***********************************
13                      ;*  Reserve 100 more bytes in var1 *
14                      ;***********************************
15 0001                 array   .usect  "var1",100
16
17 0004   50340037              add #37h, R4 ; still in .text
18
19                      ;***********************************
20                      ;*  Reserve 50 more bytes in var1  *
21                      ;***********************************
22 0065                 dflag   .usect  "var1",50
23
24 0008  -4014005b              mov dflag, R4; still in .text
25
26                      ;***********************************
27                      ;*  Reserve 100 bytes in var2      *
28                      ;***********************************
29 0000                 vec     .usect  "var2", 100
30
31 000c  -90140000              cmp vec, R4  ; still in .text
32
33                      ;***********************************
34                      ;*  Declare external .usect symbol *
35                      ;***********************************
36                             .global array
```



Example 4.8: The .usect Directive

Syntax .**word** *value*$_1$ [*, ... , value*$_n$]

Description The .word directive places one or more 16–bit values into consecutive two–byte pairs in the current section.

The *values* can be either absolute or relocatable expressions. If an expression is relocatable, the assembler generates a relocation entry that refers to the appropriate symbol; the linker can then correctly patch (relocate) the reference. This allows you to initialize memory with pointers to variables or labels.

You can use as many *values* as fit on a single line. If you use a label, it points to the first word that is initialized.

Note that when you use .word in a .struct/.endstruct sequence, it defines a member's size; it does not initialize memory.

Example This example shows how to use the .word directive to initialize words. The symbol WordX points to the first word that is reserved.

```
1 0000   0c80  Wordx:  .word   3200, 1+’B’, -0afh, ’X’
  0002   0043
  0004   ff51
  0006   0058
```