4.1 Directives Summary

Table 4-1 through Table 4-9 summarize the assembler directives.

Besides the assembler directives documented here, the MSP430™ software tools support the following directives:

- The assembler uses several directives for macros. Macro directives are discussed in Chapter 5; they are not discussed in this chapter.
- The C compiler uses directives for symbolic debugging. Unlike other directives, symbolic debugging directives are not used in most assembly language programs. Appendix A discusses these directives; they are not discussed in this chapter.

Labels and Comments Are Not Shown in Syntaxes

Note: Any source statement that contains a directive can also contain a label and a comment. Labels begin in the first column (only labels and comments can appear in the first column), and comments must be preceded by a semicolon, or an asterisk if the comment is the only element in the line. To improve readability, labels and comments are not shown as part of the directive syntax.

Table 4-1. Directives That Define Sections

Mnemonic and Syntax	Description	See
.bss *symbol, size in bytes*[, *alignment*]	Reserves *size* bytes in the .bss (uninitialized data) section	.bss topic
.data	Assembles into the .data (initialized data) section	.data topic
.sect " *section name*"	Assembles into a named (initialized) section	.sect topic
.text	Assembles into the .text (executable code) section	.text topic
symbol **.usect "***section name*", *size in bytes* [, *alignment*]	Reserves *size* bytes in a named (uninitialized) section	.usect topic

Table 4-2. Directives That Initialize Constants (Data and Memory)

Mnemonic and Syntax	Description	See
.byte $value_1$[, ..., $value_n$]	Initializes one or more successive bytes in the current section	.byte topic
.char $value_1$[, ..., $value_n$]	Initializes one or more successive bytes in the current section	.char topic
.double $value_1$[, ..., $value_n$]	Initializes one or more 32-bit, IEEE double-precision, floating-point constants	.double topic
.field *value*[, *size*]	Initializes a field of *size* bits (1-32) with *value*	.field topic
.float $value_1$[, ..., $value_n$]	Initializes one or more 32-bit, IEEE single-precision, floating-point constants	.float topic
.half $value_1$[, ... , $value_n$]	Initializes one or more 16-bit integers (halfword)	.half topic
.int $value_1$[, ... , $value_n$]	Initializes one or more 16-bit integers	.int topic
.long $value_1$[, ... , $value_n$]	Initializes one or more 32-bit integers	.long topic
.short $value_1$[, ... , $value_n$]	Initializes one or more 16-bit integers (halfword)	.short topic
.string {$expr_1$|**"**$string_1$**"**}[,... , {$expr_n$|**"**$string_n$**"**}]	Initializes one or more text strings	.string topic
.word $value_1$[, ... , $value_n$]	Initializes one or more 16-bit integers	.word topic

Table 4-3. Directives That Perform Alignment and Reserve Space

Mnemonic and Syntax	Description	See
.align [*size in bytes*]	Aligns the SPC on a boundary specified by *size in bytes*, which must be a power of 2; defaults to word (2 byte) boundary	.align topic
.bes *size*	Reserves *size* bytes in the current section; a label points to the end of the reserved space	.bes topic
.space *size*	Reserves *size* bytes in the current section; a label points to the beginning of the reserved space	.space topic

Table 4-4. Directives That Format the Output Listing

Mnemonic and Syntax	Description	See
.drlist	Enables listing of all directive lines (default)	.drlist topic
.drnolist	Suppresses listing of certain directive lines	.drnolist topic
.fclist	Allows false conditional code block listing (default)	.fclist topic
.fcnolist	Suppresses false conditional code block listing	.fcnolist topic
.length [*page length*]	Sets the page length of the source listing	.length topic
.list	Restarts the source listing	.list topic
.mlist	Allows macro listings and loop blocks (default)	.mlist topic
.mnolist	Suppresses macro listings and loop blocks	.mnolist topic
.nolist	Stops the source listing	.nolist topic
.option $option_1$**[,** $option_2$**,** . . .]	Selects output listing options; available options are A, B, H, M, N, O, R, T, W, and X	.option topic
.page	Ejects a page in the source listing	.page topic
.sslist	Allows expanded substitution symbol listing	.sslist topic
.ssnolist	Suppresses expanded substitution symbol listing (default)	.ssnolist topic
.tab *size*	Sets tab to *size* characters	.tab topic
.title " *string* **"**	Prints a title in the listing page heading	.title topic
.width [*page width*]	Sets the page width of the source listing	.width topic

Table 4-5. Directives That Reference Other Files

Mnemonic and Syntax	Description	See
.copy ["]*filename*["]	Includes source statements from another file	.copy topic
.def $symbol_1$**[,** ... **,** $symbol_n$]	Identifies one or more symbols that are defined in the current module and that can be used in other modules	.def topic
.global $symbol_1$**[,** ... **,** $symbol_n$]	Identifies one or more global (external) symbols	.global topic
.include ["]*filename*["]	Includes source statements from another file	.include topic
.mlib ["]*filename*["]	Defines macro library	.mlib topic
.ref $symbol_1$**[,** ... **,** $symbol_n$]	Identifies one or more symbols used in the current module that are defined in another module	.ref topic

Table 4-6. Directives That Enable Conditional Assembly

Mnemonic and Syntax	Description	See
.break [*well-defined expression*]	Ends .loop assembly if *well-defined expression* is true. When using the .loop construct, the .break construct is optional.	.break topic
.else	Assembles code block if the .if *well-defined expression* is false. When using the .if construct, the .else construct is optional.	.else topic
.elseif *well-defined expression*	Assembles code block if the .if *well-defined expression* is false and the .elseif condition is true. When using the .if construct, the .elseif construct is optional.	.elseif topic
.endif	Ends .if code block	.endif topic
.endloop	Ends .loop code block	.endloop topic
.if *well-defined expression*	Assembles code block if the *well-defined expression* is true	.if topic
.loop [*well-defined expression*]	Begins repeatable assembly of a code block; the loop count is determined by the *well-defined expression*.	.loop topic

Table 4-7. Directives That Define Structures

Mnemonic and Syntax	Description	See
.endstruct	Ends a structure definition	.struct
.struct	Begins structure definition	.struct topic
.tag	Assigns structure attributes to a label	.struct

Table 4-8. Directives That Define Symbols at Assembly Time

Mnemonic and Syntax	Description	See
.asg ["]*character string*["], *substitution symbol*	Assigns a character string to *substitution symbol*	.asg topic
symbol **.equ** *value*	Equates *value* with *symbol*	.equ topic
.eval *well-defined expression*, *substitution symbol*	Performs arithmetic on a numeric *substitution symbol*	.eval topic
.label *symbol*	Defines a load-time relocatable label in a section	.label topic
symbol **.set** *value*	Equates *value* with *symbol*	.set topic
.var	Adds a local substitution symbol to a macro's parameter list	.var topic

Table 4-9. Directives That Perform Miscellaneous Functions

Mnemonic and Syntax	Description	See
.asmfunc	Identifies the beginning of a block of code that contains a function	.asmfunc topic
.cdecls [*options*,] **"***filename***"**[, **"**filename2**"**[, ...]	Share C headers between C and assembly code	.cdecls topic
.clink [**"***section name***"**]	Enables conditional linking for the current or specified section	.clink topic
.emsg *string*	Sends user-defined error messages to the output device; produces no .obj file	.emsg topic
.end	Ends program	.end topic
.endasmfunc	Identifies the end of a block of code that contains a function	.endasmfunc topic
.mmsg *string*	Sends user-defined messages to the output device	.mmsg topic
.newblock	Undefines local labels	.newblock topic
.wmsg *string*	Sends user-defined warning messages to the output device	.wmsg topic

4.2 Directives That Define Sections

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

- The **.bss** directive reserves space in the .bss section for uninitialized variables.
- The **.data** directive identifies portions of code in the .data section. The .data section usually contains initialized data.
- The **.sect** directive defines an initialized named section and associates subsequent code or data with that section. A section defined with .sect can contain code or data.
- The **.text** directive identifies portions of code in the .text section. The .text section usually contains executable code.
- The **.usect** directive 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.

Chapter 2 discusses these sections in detail.

Example 4-1 shows how you can use sections directives to associate code and data with the proper sections. This is an output listing; 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 after other code is assembled, the section's SPC resumes counting as if there had been no intervening code.

The directives in Example 4-1 perform the following tasks:

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

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.

Example 4-1. Sections Directives

```
 1            ; Comment1 here : Start assembling ....
 2
 3 0000               .text
 4 0000 0001          .word   1,2
   0002 0002
 5 0004 0003          .word   3,4
   0006 0004
 6
 7            ; Comment 2
 8
 9 0000               .data
10 0000 0009          .word   9,10
   0002 000A
11 0004 000B          .word   11,12
   0006 000C
12
13            ; Comment 3
14
15 0000               .sect   "var_defs"
16 0000 0011          .word   17,18
  0002 0012
17
18            ; Comment 4
19
20 0008               .data
21 0008 000D          .word   13,14
   000a 000E
22
23 0000               .bss    sym,19
24 000c 000F          .word   15,16
   000e 0010
25
26            ; Comment 5
27
28 0008               .text
29 0008 0005          .word   5,6
   000a 0006
30 0000 usym          .usect  "xy",20
31 000c 0007          .word   7,8
   000e 0008
```

4.3 Directives That Initialize Constants

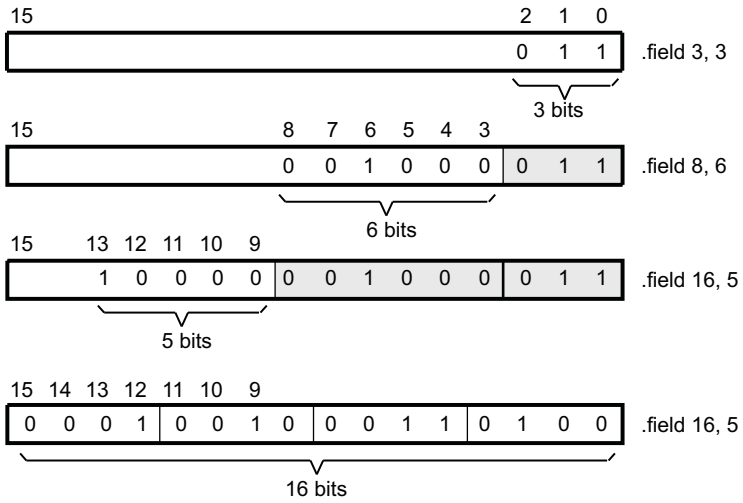
Several directives assemble values for the current section:

- The **.byte** and **.char** directives place one or more 8-bit values into consecutive bytes of the current section. These directives are similar to .long and .word, except that the width of each value is restricted to eight bits.
- The **double** and **.float** directives calculate the single-precision (32-bit) IEEE floating-point representation of a single floating-point value and store it in a word in the current section that is aligned to a word boundary.
- The **.field** directive places a single value into a specified number of bits in the current word. With .field, you can pack multiple fields into a single word; the assembler does not increment the SPC until a word is filled.

  Figure 4-1 shows how fields are packed into a word. Using the following assembled code, notice that the SPC does not change for the first three fields (the fields are packed into the same word).

```
1 0000 0003             .field 3,3
2 0000 0043             .field 8,6
3 0000 2043             .field 16,5
4 0002 1234             .field 0x1234,16
```

Figure 4-1. The .field Directive



- The **.int** and **.word** directives place one or more 16-bit values into consecutive words in the current section.
- The **.string** directive places 8-bit characters from one or more character strings into the current section. This directive is similar to .byte, placing an 8-bit character in each consecutive byte of the current section.
- The **.long** directive places one or more 32-bit values into consecutive words in the current section.

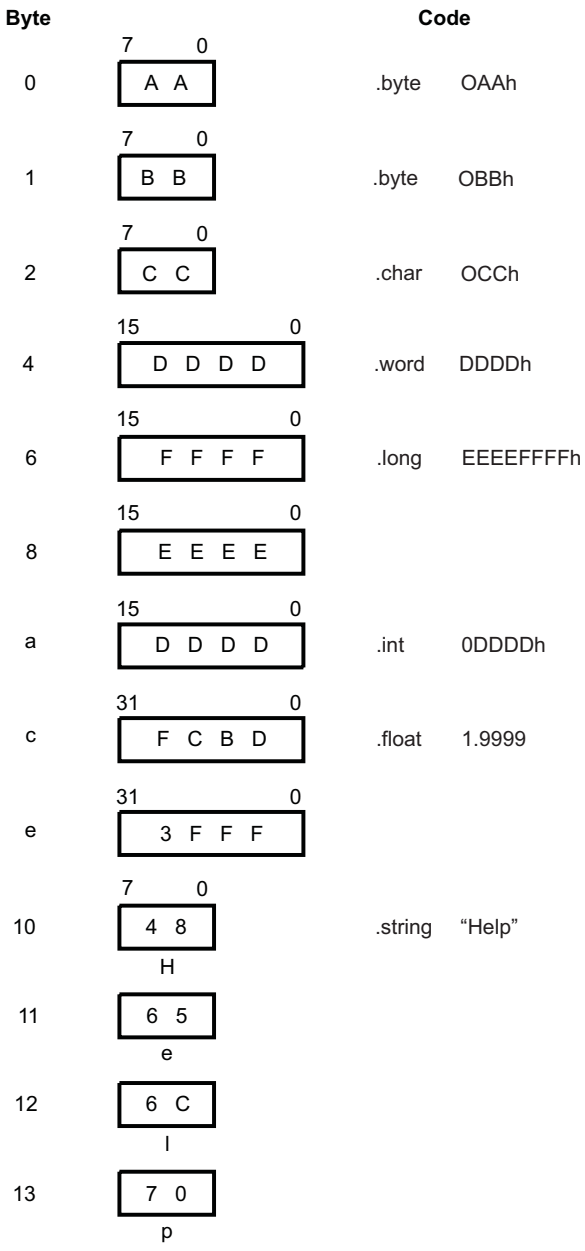
Directives That Initialize Constants When Used in a .struct/.endstruct Sequence

Note: The .byte, .char, .word, .int, .long, .string, .double, .float, .half, .short, and .field directives do not initialize memory when they are part of a .struct/ .endstruct sequence; rather, they define a member's size. For more information, see the .struct/.endstruct directives .

Figure 4-2 compares the .byte, .char, .int, .long, .float, .word, and .string directives, using the following assembled code:

```
1 0000 00AA          .byte   0AAh,0BBh
  0001 00BB
2 0002 00CC          .char   0xCC
3 0004 DDDD          .word   0xDDDD
4 0006 FFFF          .long   0xEEEEFFFF
  0008 EEEE
5 000a DDDD          .int    0xDDDD
6 000c FCB9          .float  1.9999
  000e 3FFF
7 0010 0048          .string "Hello"
  0011 0065
  0012 006C
  0013 0070
```

Figure 4-2. Initialization Directives



4.4 Directives That Perform Alignment and Reserve Space

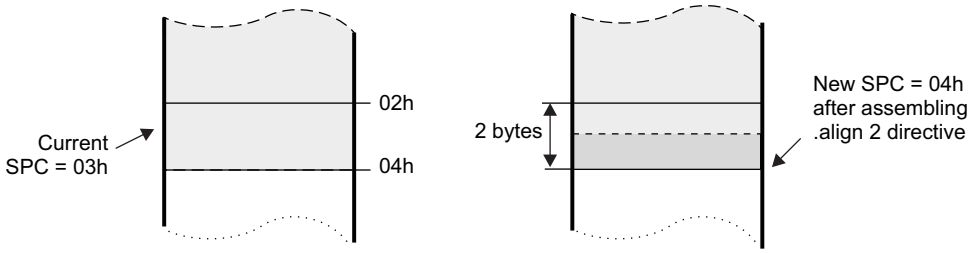
These directives align the section program counter (SPC) or reserve space in a section:

- The **.align** directive at a 1-byte to 32K-byte boundary. This ensures that the code following the directive begins on the byte value that you specify. If the SPC is already aligned at the selected boundary, it is not incremented. Operands for the .align directive must equal a power of 2 between 20 and 215, inclusive. The .align directive with no operands defaults to 2, that is, to a word boundary.

  Figure 4-3 demonstrates the .align directive. Using the following assembled code:

```
1 0000 0002          .field  2,3
2 0000 005A          .field  11,5
3                    .align  2
4 0002 0045          .string "Err"
  0003 0072
  0004 0072
5                    .align
6 0006 0004          .byte   4
```

Figure 4-3. The .align Directive



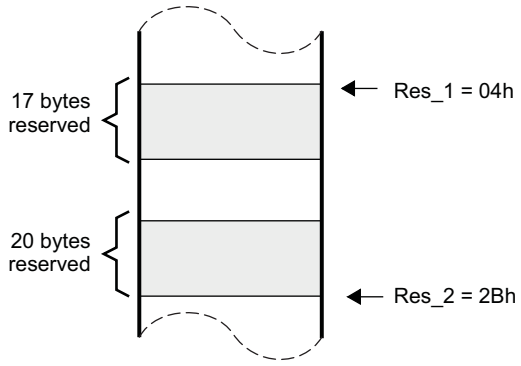
- The **.bes** and **.space** directives reserve a specified number of bytes in the current section. The assembler fills these reserved bytes with 0s.
  - When you use a label with .space, it points to the *first* byte that contains reserved bits.
  - When you use a label with .bes, it points to the *last* byte that contains reserved bits.

  Figure 4-4 shows how the .space and .bes directives work for the following assembled code:

```
1 0000 0100           .word  0x100,0x200
  0002 0200
2 0004       Res_1    .space 17
3 0016 000F           .word  15
4 002b       Res_2    .bes   20
5 002c 00BA           .byte  0xBA
```

Res_1 points to the first byte in the space reserved by .space. Res_2 points to the last byte in the space reserved by .bes.

Figure 4-4. The .space and .bes Directives



4.5 Directives That Format the Output Listings

These directives format the listing file:

- The **.drlist** directive causes printing of the directive lines to the listing; the **.drnolist** directive turns it off for certain directives. You can use the .drnolist directive to suppress the printing of the following directives. You can use the .drlist directive to turn the listing on again.

.asg	.eval	.length	.mnolist	.var
.break	.fclist	.mlist	.sslist	.width
.emsg	.fcnolist	.mmsg	.ssnolist	.wmsg

- The source code listing includes 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 **.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 on again.
- The source code listing includes macro expansions and loop blocks. The **.mlist** and **.mnolist** directives turn this listing on and off. You can use the .mlist directive to print all macro expansions and loop blocks to the listing, and the .mnolist directive to suppress this listing.
- The **.option** directive controls certain features in the listing file. This directive has the following operands:

  A turns on listing of all directives and data, and subsequent expansions, macros, and blocks.

  B limits the listing of .byte and .char directives to one line.

  H limits the listing of .half and .short directives to one line.

  M turns off macro expansions in the listing.

  N turns off listing (performs .nolist).

  O turns on listing (performs .list).

  R resets the B, H, M, T, and W directives (turns off the limits of B, H, M, T, and W).

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

  W limits the listing of .word and .int 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 --cross_reference option (see Section 3.3).

- The **.page** directive causes a page eject in the output listing.
- The source code listing includes substitution symbol expansions. The **.sslist** and **.ssnolist** directives turn this listing on and off. You can use the .sslist directive to print all substitution symbol expansions to the listing, and the .ssnolist directive to suppress this listing. These directives are useful for debugging the expansion of substitution symbols.
- The **.tab** directive defines tab size.
- The **.title** directive supplies a title that the assembler prints at the top of each page.
- The **.width** directive controls the page width of the listing file. You can use this directive to adjust listings for various output devices.

4.6 Directives That Reference Other Files

These directives supply information for or about other files that can be used in the assembly of the current file:

- 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 **.def** directive identifies a symbol that is defined in the current module and that can be used in another module. The assembler includes the symbol in the symbol table.
- The **.global** directive declares a symbol external so that it is available to other modules at link time. (For more information about global symbols, see Section 2.7.1). The .global directive does double duty, acting as a .def for defined symbols and as a .ref for undefined symbols. The linker resolves an undefined global symbol reference only if the symbol is used in the program. The .global directive declares a 16-bit symbol.
- 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.
- The **.ref** directive identifies a symbol that is used in the current module but is defined in another module. The assembler marks the symbol as an undefined external symbol and enters it in the object symbol table so the linker can resolve its definition. The .ref directive forces the linker to resolve a symbol reference.

4.7 Directives That Enable Conditional Assembly

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 *well-defined expression*	marks the beginning of a conditional block and assembles code if the .if *well-defined expression* is true.
[**.elseif** *well-defined expression*]	marks a block of code to be assembled if the .if *well-defined expression* is false and the .elseif condition is true.
.else	marks a block of code to be assembled if the .if *well-defined expression* is false and any .elseif conditions are 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 [*well-defined expression*]	marks the beginning of a repeatable block of code. The optional expression evaluates to the loop count.
.break [*well-defined expression*]	tells the assembler to assemble repeatedly when the .break *well-defined expression* is false and to go to the code immediately after .endloop when the expression is true or omitted.
.endloop	marks the end of a repeatable block.

The assembler supports several relational operators that are useful for conditional expressions. For more information about relational operators, see Section 3.9.4.

4.8 Directives That Define Structures

The **.struct/.endstruct** directives set up C-like structure definitions. The **.union/.endunion** directives set up C-like union definitions. The **.tag** directive assigns the C-like structure or union characteristics to a label.

The .struct/.endstruct directives allow you to organize your information into structures so that similar elements can be grouped together. Similarly, the .union/.endunion directives allow you to organize your information into unions. Element offset calculation is left up to the assembler. These directives do not allocate memory. They simply create a symbolic template that can be used repeatedly.

The .tag directive assigns a label to a structure or union. 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     .int
Y     .int
T_LEN .endstruct

COORD .tag type        ; declare COORD (coordinate)

COORD .space T_LEN     ; actual memory allocation
LDR   R0, COORD.Y      ; load member Y of structure
                       ; COORD into register R0.
```

4.9 Directives That Define Symbols at Assembly Time

Assembly-time symbol directives equate meaningful symbol names to constant values or strings.

- 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
        ; Assign string to substitution symbol.
.byte coefficients
        ; Place the symbol values 10, 20, 30, and 40
        ; into consecutive bytes in current section.
```

- The **.eval** directive evaluates a well-defined 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:

```
.asg      1 , x   ; x = 1
.loop             ; Begin conditional loop.
.byte     x*10h   ; Store value into current section.
.break    x = 4   ; Break loop if x = 4.
.eval     x+1, x  ; Increment x by 1.
.endloop          ; End conditional loop.
```

- The **.label** directive defines a special symbol that refers to the load-time address within the current section. This is useful when a section loads at one address but runs at a different address. For example, you may want to load a block of performance-critical code into slower off-chip memory to save space and move the code to high-speed on-chip memory to run. See the .label topic for an example using a load-time address label.
- 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   ; Set bval = 0100h
     .long bval, bval*2, bval+12
        ; Store the values 0100h, 0200h, and 010Ch
        ; into consecutive words in current section.
```

  The .set and .equ directives produce no object code. The two directives are identical and can be used interchangeably.
- The **.var** directive allows you to use substitution symbols as local variables within a macro.

4.10 Miscellaneous Directives

These directives enable miscellaneous functions or features:

- The **.asmfunc** and **.endasmfunc** directives mark function boundaries. These directives are used with the compiler --symdebug:dwarf (-g) option to generate debug information for assembly functions.
- The **.cdecls** directive enables programmers in mixed assembly and C/C++ environments to share C headers containing declarations and prototypes between C and assembly code.
- The **.clink** directive enables conditional linking by telling the linker to leave the named section out of the final object module output of the linker if there are no references found to any symbol in the section. The .clink directive can be applied to initialized or uninitialized sections.
- The **.end** directive terminates assembly. If you use the .end directive, it should be the last source statement of a program. This directive has the same effect as an end-of-file character.
- The **.newblock** directive resets local labels. Local labels are symbols of the form $n, where n is a decimal digit. They are defined when they appear in the label field. Local labels are temporary labels that can be used as operands for jump instructions. The .newblock directive limits the scope of local labels by resetting them after they are used. See Section 3.8.2 for information on local labels.

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 same manner as the assembler, incrementing the error count and preventing 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 manner as the .emsg and .wmsg directives but does not set the error count or the warning count. It does not affect the creation of the object file.
- The **.wmsg** directive sends warning messages to the standard output device. The .wmsg directive functions in the same manner as the .emsg directive but increments the warning count rather than the error count. It does not affect the creation of the object file.

4.11 Directives Reference

The remainder of this chapter is a reference. Generally, the directives are organized alphabetically, one directive per topic. Related directives (such as .if/.else/.endif), however, are presented together in one topic.

.align *Align SPC on the Next Boundary*

Syntax

.align [*size in bytes*]

Description

The .align directive aligns the section program counter (SPC) on the next boundary, depending on the size in bytes parameter. The size may be any power of 2 between 2^0 and 2^{15}, inclusive. An operand of 2 aligns the SPC on the next word boundary, and this is the default if no size is given. For example:

2	aligns SPC to word boundary
4	aligns SPC to 2 word boundary
128	aligns SPC to 128-byte boundary

Using the .align directive has two effects:

- The assembler aligns the SPC on a byte value that you specify *within* the current section.
- The assembler sets a flag that forces the linker to align the section so that individual alignments remain intact when a section is loaded into memory.

Example

This example shows several types of alignment, including .align 4, .align 8, and a default .align.

```
 1 0000 0004       .byte 4
 2                 .align 2
 3 0002 0045       .string "Errorcnt"
   0003 0072
   0004 0072
   0005 006F
   0006 0072
   0007 0063
   0008 006E
   0009 0074
 4                 .align
 5 000a 0003       .field 3,3
 6 000a 002B       .field 5,4
 7                 .align 2
 8 000c 0003       .field 3,3
 9                 .align 8
10 0010 0005       .field 5,4
11                 .align
12 0012 0004       .byte 4
```

.asg/.eval *Assign a Substitution Symbol*

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 (which cannot be redefined) to a symbol, .asg assigns a character string (which 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* must be a valid symbol name. The substitution symbol is up to 128 characters long and must begin with a letter. Remaining characters of the symbol can be a combination of alphanumeric characters, the underscore (_), and the dollar sign ($).

The **.eval** directive performs arithmetic on substitution symbols, which are stored in the substitution symbol table. This directive evaluates the *well-defined 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.

- The *well-defined expression* is an alphanumeric expression in which all symbols have been previously defined in the current source module, so that the result is an absolute.
- The *substitution symbol* must be a valid symbol name. The substitution symbol is up to 128 characters long and must begin with a letter. Remaining characters of the symbol can be a combination of alphanumeric characters, the underscore (_), and the dollar sign ($).

Example

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

```
      1                    .sslist         ; show expanded sub. symbols
      2                                    ; using .asg and .eval
      3
      4                    .asg    R12, LOCALSTACK
      5                    .asg    &, AND
      6
      7 0000 503C          ADD     #280 AND 255, LOCALSTACK
#                          ADD     #280 & 255, R12
        0002 0018
      8
      9                    .asg    0,x
     10                    .loop   5
     11                    .eval   x+1, x
     12                    .word   x
     13                    .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
```

.asmfunc/.endasmfunc *Mark Function Boundaries*

Syntax

symbol **.asmfunc**

.endasmfunc

Description

The **.asmfunc** and **.endasmfunc** directives mark function boundaries. These directives are used with the compiler -g option (--symdebug:dwarf) to allow assembly code sections to be debugged in the same manner as C/C++ functions.

You should not use the same directives generated by the compiler (see Appendix A) to accomplish assembly debugging; those directives should be used only by the compiler to generate symbolic debugging information for C/C++ source files.

The .asmfunc and .endasmfunc directives cannot be used when invoking the compiler with the backwards-compatibility --symdebug:coff option. This option instructs the compiler to use the obsolete COFF symbolic debugging format, which does not support these directives.

The *symbol* is a label that must appear in the label field.

Consecutive ranges of assembly code that are not enclosed within a pair of .asmfunc and .endasmfunc directives are given a default name in the following format:

$ *filename* **:** *beginning source line* **:** *ending source line* **$**

.bss *Reserve Space in the .bss Section*

Syntax

.bss *symbol*, *size in bytes*[, *alignment*]

Description

The **.bss** directive reserves space for variables in the .bss section. This directive is usually used to allocate space in RAM.

- The *symbol* is a required parameter. It defines a label that points to the first location reserved by the directive. The symbol name must correspond to the variable that you are reserving space for.
- The *size in bytes* is a required parameter; it must be an absolute expression. The assembler allocates size bytes in the .bss section. There is no default size.
- The *alignment* is an optional parameter that ensures that the space allocated to the symbol occurs on the specified boundary. The boundary indicates the size of the alignment in bytes and must be set to a power of 2 between 20 and 215, inclusive. If the SPC is aligned at the specified boundary, it is not incremented.

For more information about sections, see Chapter 2.

Example

In this example, the .bss directive allocates space for two variables, TEMP and ARRAY. The symbol TEMP points to four bytes of uninitialized space (at .bss SPC = 0). The symbol ARRAY points to 100 bytes of uninitialized space (at .bss SPC = 04h). Symbols declared with the .bss directive can be referenced in the same manner as other symbols and can also be declared external.

```
 1                ***********************************************
 2                **  Start assembling into the .text section. **
 3                ***********************************************
 4 0000                  .text
 5 0000 430A             MOV     #0, R10
 6
 7                ***********************************************
 8                **     Allocate 4 bytes in .bss for TEMP.    **
 9                ***********************************************
10 0000           Var_1:  .bss    TEMP, 4
11
12                ***********************************************
13                **           Still in .text.                 **
14                ***********************************************
15 0002 503B             ADD     #56h, R11
   0004 0056
16 0006 5C0B             ADD     R12, R11
17
18                ***********************************************
19                ** Allocate 100 bytes in .bss for the symbol **
20                **                named ARRAY.               **
21                ***********************************************
22 0004                  .bss    ARRAY, 100, 4
23
24                ***********************************************
25                **       Assemble more code into .text.      **
26                ***********************************************
27 0008 4130             RET
28
29                ***********************************************
30                **      Declare external .bss symbols.       **
31                ***********************************************
32                       .global ARRAY, TEMP
33                       .end
```

.byte/.char *Initialize Byte*

Syntax

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

.char *value*$_1$[**,** ... **,** *value*$_n$]

Description

The **.byte** and **.char** directives place one or more values into consecutive bytes of the current section. A *value* can be one of the following:

- An expression that the assembler evaluates and treats as an 8-bit signed number
- A character string enclosed in double quotes. Each character in a string represents a separate value, and values are stored in consecutive bytes. The entire string *must* be enclosed in quotes.

The assembler truncates values greater than eight 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 where the assembler places the first byte.

When you use .byte or .char in a .struct/.endstruct sequence, .byte and .char define a member's size; they do not initialize memory. For more information, see the .struct/.endstruct/.tag topic .

Example

In this example, 8-bit values (10, -1, abc, and a) are placed into consecutive bytes in memory with .byte. Also, 8-bit values (8, -3, def, and b) are placed into consecutive bytes in memory with .char. 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 .char directive.

```
1 0000                .space  100h
2 0100 000A  STRX     .byte   10, -1, "abc", 'a'
  0101 00FF
  0102 0061
  0103 0062
  0104 0063
  0105 0061
3 0106 0008           .char   8, -3, "def", 'b'
  0107 00FD
  0108 0064
  0109 0065
  010a 0066
  010b 0062
```

.cdecls *Share C Headers Between C and Assembly Code*

Syntax **Single Line:**

.cdecls [*options*,] **"***filename***"[, "***filename2***"[,...]]**

Syntax **Multiple Lines:**

```
.cdecls [options]
%{
/*----------------------------------------------------------------------------*/
/* C/C++ code - Typically a list of #includes and a few defines */
/*----------------------------------------------------------------------------*/
%}
```

Description

The **.cdecls** directive allows programmers in mixed assembly and C/C++ environments to share C headers containing declarations and prototypes between the C and assembly code. Any legal C/C++ can be used in a .cdecls block and the C/C++ declarations cause suitable assembly to be generated automatically, allowing you to reference the C/C++ constructs in assembly code; such as calling functions, allocating space, and accessing structure members; using the equivalent assembly mechanisms. While function and variable definitions are ignored, most common C/C++ elements are converted to assembly, for instance: enumerations, (non-function-like) macros, function and variable prototypes, structures, and unions.

The .cdecls options control whether the code is treated as C or C++ code; and how the .cdecls block and converted code are presented. Options must be separated by commas; they can appear in any order:

C	Treat the code in the .cdecls block as C source code (default).
CPP	Treat the code in the .cdecls block as C++ source code. This is the opposite of the C option.
NOLIST	Do not include the converted assembly code in any listing file generated for the containing assembly file (default).
LIST	Include the converted assembly code in any listing file generated for the containing assembly file. This is the opposite of the NOLIST option.
NOWARN	Do not emit warnings on STDERR about C/C++ constructs that cannot be converted while parsing the .cdecls source block (default).
WARN	Generate warnings on STDERR about C/C++ constructs that cannot be converted while parsing the .cdecls source block. This is the opposite of the NOWARN option.

In the single-line format, the options are followed by one or more filenames to include. The filenames and options are separated by commas. Each file listed acts as if #include "filename" was specified in the multiple-line format.

In the multiple-line format, the line following .cdecls must contain the opening .cdecls block indicator %{. Everything after the %{, up to the closing block indicator %}, is treated as C/C++ source and processed. Ordinary assembler processing then resumes on the line following the closing %}.

The text within %{ and %} is passed to the C/C++ compiler to be converted into assembly language. Much of C language syntax, including function and variable definitions as well as function-like macros, is not supported and is ignored during the conversion. However, all of what traditionally appears in C header files is supported, including function and variable prototypes; structure and union declarations; non-function-like macros; enumerations; and #define's.

The resulting assembly language is included in the assembly file at the point of the .cdecls directive. If the LIST option is used, the converted assembly statements are printed in the listing file.

The assembly resulting from the .cdecls directive is treated similarly to a .include file. Therefore the .cdecls directive can be nested within a file being copied or included. The assembler limits nesting to ten 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.

The .cdecls directive can appear anywhere in an assembly source file, and can occur multiple times within a file. However, the C/C++ environment created by one .cdecls is **not** inherited by a later .cdecls; the C/C++ environment starts new for each .cdecls.

See Chapter 12 for more information on setting up and using the .cdecls directive with C header files.

Example In this example, the .cdecls directive is used call the C header.h file.

C header file:

```
#define WANT_ID 10
#define NAME "John\n"

extern int a_variable;
extern float cvt_integer(int src);

struct myCstruct { int member_a; float member_b; };

enum status_enum { OK = 1, FAILED = 256, RUNNING = 0 };
```

Source file:

```
         .cdecls C,LIST,"myheader.h"

size:    .int $sizeof(myCstruct)
aoffset: .int myCstruct.member_a
boffset: .int myCstruct.member_b
okvalue: .int status_enum.OK
failval: .int status_enum.FAILED
         .if $$defined(WANT_ID)
id       .cstring NAME
         .endif
```

Listing File:

```
      1                    .cdecls C,LIST,"myheader.h"
A     1            ; ----------------------------------------
A     2            ; Assembly Generated from C/C++ Source Code
A     3            ; ----------------------------------------
A     4
A     5            ; =========== MACRO DEFINITIONS ===========
A     6                      .define "1",WANT_ID
A     7                      .define """John\n""",NAME
A     8                      .define "1",_OPTIMIZE_FOR_SPACE
A     9
A    10            ; =========== TYPE DEFINITIONS ===========
A    11            status_enum      .enum
A    12      0001  OK               .emember 1
A    13      0100  FAILED           .emember 256
A    14      0000  RUNNING          .emember 0
A    15                             .endenum
A    16
A    17            myCstruct        .struct 0,2          ; struct size=(6 bytes|48 bits), alignment=2
A    18      0000  member_a         .field 16            ; int member_a - offset 0 bytes, size (2
bytes|16 bits)
A    19      0002  member_b         .field 32            ; float member_b - offset 2 bytes, size (4
bytes|32 bits)
```

```
A    20      0006                    .endstruct          ; final size=(6 bytes|48 bits)
A    21
A    22           ; =========== EXTERNAL FUNCTIONS ===========
A    23                               .global cvt_integer
A    24
A    25           ; =========== EXTERNAL VARIABLES ===========
A    26                               .global a_variable
      2
      3 0000 0006  size:    .int $sizeof(myCstruct)
      4 0002 0000  aoffset: .int myCstruct.member_a
      5 0004 0002  boffset: .int myCstruct.member_b
      6 0006 0001  okvalue: .int status_enum.OK
      7 0008 0100  failval: .int status_enum.FAILED
      8                     .if $defined(WANT_ID)
      9 000a 004A  id       .cstring NAME
        000b 006F
        000c 0068
        000d 006E
        000e 000A
        000f 0000
     10                     .endif
```

.clink *Conditionally Leave Section Out of Object Module Output*

Syntax

.clink ["*section name*"]

Description

The **.clink** directive enables conditional linking by telling the linker to leave a section out of the final object module output of the linker if there are no references found to any symbol in *section name*. The .clink directive can be applied to initialized or uninitialized sections.

The *section name* identifies the section. If .clink is used without a section name, it applies to the current initialized section. If .clink is applied to an uninitialized section, the section name is required. The section name is significant to 200 characters and must be enclosed in double quotes. A section name can contain a subsection name in the form *section name*:*subsection name*.

The .clink directive tells the linker to leave the section out of the final object module output of the linker if there are no references found in a linked section to any symbol defined in the specified section. The --absolute_exe linker option produces the final output in the form of an absolute, executable output module.

A section in which the entry point of a C program is defined cannot be marked as a conditionally linked section.

Example

In this example, the Vars and Counts sections are set for conditional linking.

```
 1            ************************************************
 2            ** Set Vars section for conditional linking.   **
 3            ************************************************
 4 0000               .sect "Vars"
 5                    .clink
 6 0000 00AA  X:      .word 0AAh
 7 0002 00AA  Y:      .word 0AAh
 8 0004 00AA  Z:      .word 0AAh
 9            ************************************************
10            ** Set Counts section for conditional linking. **
11            ************************************************
12 0000               .sect "Counts"
13                    .clink
14 0000 00AA  XCount: .word 0AAh
15 0002 00AA  YCount: .word 0AAh
16 0004 00AA  ZCount: .word 0AAh
17            ************************************************
18            ** .text is unconditionally linked by default. **
19            ************************************************
20 0000               .text
21 0000 403B          MOV  #X_addr, R11
   0002 0008!
22 0004 4B2B          MOV  @R11, R11
23 0006 5B0C          ADD  R11, R12
24
25 0008 0000! X_addr: .field  X, 32
   000a 0000
26            ************************************************
27            ** The reference to symbol X causes the Vars   **
28            ** section to be linked into the COFF output.  **
29            ************************************************
```

.copy/.include *Copy Source File*

Syntax

.copy ["]*filename*["]

.include ["]*filename*["]

Description

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

When a .copy or .include directive is assembled, 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. It can be enclosed in double quotes and must follow operating system conventions. If *filename* starts with a number the double quotes are required.

You can specify a full pathname (for example, /320tools/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 --include_path assembler option
3. Any directories specified by the MSP430_A_DIR environment variable
4. Any directories specified by the MSP430_C_DIR environment variable

For more information about the --include_path option and MSP430_A_DIR, see Section 3.4. For more information about MSP430_C_DIR, see the *MSP430 Optimizing C/C++ Compiler User's Guide*.

The .copy and .include directives can be nested within a file being copied or included. The assembler limits nesting to 32 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. A indicates the first copied file, B indicates a second copied file, etc.

Example 1

In this example, the .copy directive is used to read and assemble source statements from other files; then, the assembler resumes assembling into the current file.

The original file, copy.asm, contains a .copy statement copying the file byte.asm. When copy.asm assembles, the assembler copies byte.asm into its place in the listing (note listing below). The copy file byte.asm contains a .copy statement for a second file, word.asm.

When it encounters the .copy statement for word.asm, the assembler switches to word.asm to continue copying and assembling. Then the assembler returns to its place in byte.asm to continue copying and assembling. After completing assembly of byte.asm, the assembler returns to copy.asm to assemble its remaining statement.

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 0020      .byte 32,1+ 'A'
        001e 0042
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 006A      .byte 67h + 3q
      3
      4              ** Back in original file
      5 0025 0064      .string "done"
        0026 006F
        0027 006E
        0028 0065
```

Example 2

In this example, the .include directive is used to read and assemble source statements from other files; then, the assembler resumes assembling into the current file. The mechanism is similar to the .copy directive, except that statements are not printed in the listing file.

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

`.space 29`
`.include "byte2.asm"`
`** Back in original file`
`.string "done"`
	`** In byte2.asm`
	`.byte 32,1+ 'A'`
	`.include "word2.asm"`
	`** Back in byte2.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 0064      .string "done"
        0026 006F
        0027 006E
        0028 0065
```

.data *Assemble Into the .data Section*

Syntax

.data

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.

For more information about sections, see Chapter 2.

Example

In this example, code is assembled into the .data and .text sections.

```
 1              ; Comments here
 2 0000                 .data
 3 0000                 .space 0xCC
 4
 5
 6              ; Comments here
 7 0000                 .text
 8 0000         INDEX   .set 0
 9 0000 430B            MOV #INDEX,R11
10
11
12
13              ; Comments here
14 00cc         Table:  .data
15 00cc FFFF           .word -1
16 00ce 00FF           .byte 0xFF
17
18
19
20              ; Comments here
21 0002                .text
22 0002 00CC! con      .field  Table,16
23 0004 421B           MOV     &con,R11
   0006 0002!
24 0008 5B1C           ADD     0(R11),R12
   000a 0000
25
26 00cf                .data
```

.double/.float *Initialize Single-Precision Floating-Point Value*

Syntax

.double $value_1$ [**,** ... **,** $value_n$]

.float *value* [**,** ...,$value_n$]

Description

The **.double** and **.float** directives place the IEEE single-precision floating-point representation of one or more floating-point values into the current section. Each *value* must be a floating-point constant or a symbol that has been equated to a floating-point constant. Each constant is converted to a floating-point value in IEEE single-precision 32-bit format. Floating point constants are aligned on word boundaries.

The 32-bit value is stored exponent byte first, most significant byte of fraction second, and least significant byte of fraction third, in the format shown in Figure 4-5.

Figure 4-5. Single-Precision Floating-Point Format

value = $(-1)^S$ x (1.0 + mantissa) x $(2)^{exponent-127}$

Legend: S = sign (1 bit)
E = exponent (8-bit biased)
M = mantissa (23-bit fraction)

When you use .double or .float in a .struct/.endstruct sequence, they define a member's size; they do not initialize memory. For more information, see the .struct/.endstruct/.tag topic .

Example

Following are examples of the .float and .double directives:

```
1 0000 5951     .double  -2.0e25
  0002 E984
2 0004 5951     .float   -1.0e25
  0006 E904
3 0008 0000     .double  6
  000a 40C0
4 000c 0000     .float   3
  000e 4040
```

.drlist/.drnolist *Control Listing of Directives*

Syntax

.drlist

.drnolist

Description

Two directives enable you to control the printing of assembler directives to the listing file:

The **.drlist** directive enables the printing of all directives to the listing file.

The **.drnolist** directive suppresses the printing of the following directives to the listing file. The .drnolist directive has no affect within macros.

- .asg
- .break
- .emsg
- .eval
- .fclist
- .fcnolist
- .mlist
- .mmsg
- .mnolist
- .sslist
- .ssnolist
- .var
- .wmsg

By default, the assembler acts as if the .drlist directive had been specified.

Example

This example shows how .drnolist inhibits the listing of the specified directives.

Source file:

```
.asg    0, x
.loop   2
.eval   x+1, x
.endloop
.drnolist
.asg    1, x
.loop   3
.eval   x+1, x
.endloop
```

Listing file:

```
      1                    .asg    0, x
      2                    .loop   2
      3                    .eval   x+1, x
      4                    .endloop
1                          .eval   0+1, x
1                          .eval   1+1, x
      5
      6                    .drnolist
      7
      9                    .loop   3
     10                    .eval   x+1, x
     11                    .endloop
```

.emsg/.mmsg/.wmsg *Define Messages*

Syntax

.emsg *string*

.mmsg *string*

.wmsg *string*

Description

These directives allow you to define your own error and warning messages. When you use these directives, 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 an error message to the standard output device in the same manner as the assembler. It increments the error count and prevents the assembler from producing an object file.

The **.mmsg** directive sends an assembly-time message to the standard output device in the same manner as the .emsg and .wmsg directives. It does not, however, set the error or warning counts, and it does not prevent the assembler from producing an object file.

The **.wmsg** directive sends a warning message to the standard output device in the same manner as the .emsg directive. It increments the warning count rather than the error count, however. It does not prevent the assembler from producing an object file.

Example

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

Source file:

```
MSG_EX   .macro parm1
         .if    $$symlen(parm1) = 0
         .emsg  "ERROR -- MISSING PARAMETER"
         .else
          add   parm1, r7, r8
         .endif
         .endm
         MSG_EX R11
         MSG_EX
```

Listing file:

```
       1            MSG_EX   .macro parm1
       2                     .if    $symlen(parm1)=0
       3                     .emsg  "ERROR -- MISSING PARAMETER"
       4                     .else
       5                      add   parm1, r7
       6                     .endif
       7                     .endm
       8
       9 0000                MSG_EX R11
1                            .if    $symlen(parm1)=0
1                            .emsg  "ERROR -- MISSING PARAMETER"
1                            .else
1        0000 5B07            add   R11, r7
1                            .endif
      10
      11 0002                MSG_EX
1                            .if    $symlen(parm1)=0
1                            .emsg  "ERROR -- MISSING PARAMETER"
 "emsg.asm", ERROR!   at line 11: [ ***** USER ERROR ***** - ] ERROR --

MISSING PARAMETER
1                            .else
1                             add   parm1, r7
1                            .endif

1 Assembly Error, No Assembly Warnings
```

In addition, the following messages are sent to standard output by the assembler:

```
*** ERROR!   line 11: ***** USER ERROR ***** - : ERROR -- MISSING PARAMETER
               .emsg "ERROR -- MISSING PARAMETER"
```

```
 1 Error, No Warnings
Errors in source - Assembler Aborted
```

.end *End Assembly*

Syntax **.end**

Description The **.end** directive is optional and terminates assembly. The assembler ignores any source statements that follow a .end directive. If you use the .end directive, it must be the last source statement of a program.

This directive has the same effect as an end-of-file character. You can use .end when you are debugging and you want to stop assembling at a specific point in your code.

Ending a Macro

Note: 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.

Source file:

```
START:  .space 300
TEMP    .set   15
        .bss   LOC1,0x48
LOCL_n  .word  LOC1
        MOV    #TEMP,R11
        MOV    &LOCL_n,R12
        MOV    0(R12),R13
        .end
        .byte  4
        .word  0xCCC
```

Listing file:

```
       1 0000       START:  .space 300
       2      000F  TEMP    .set   15
       3 0000               .bss   LOC1,0x48
       4 012c 0000! LOCL_n  .word  LOC1
       5 012e 403B          MOV    #TEMP,R11
         0130 000F
       6 0132 421C          MOV    &LOCL_n,R12
         0134 012C!
       7 0136 4C1D          MOV    0(R12),R13
         0138 0000
       8                    .end
```

.fclist/.fcnolist *Control Listing of False Conditional Blocks*

Syntax

.fclist

.fcnolist

Description

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

The **.fclist** directive allows the listing of false conditional blocks (conditional blocks that do not produce code).

The **.fcnolist** directive suppresses the listing of false conditional blocks until a .fclist directive is encountered. With .fcnolist, only code in conditional blocks that are actually assembled appears in the listing. The .if, .elseif, .else, and .endif directives do not appear.

By default, all conditional blocks are listed; the assembler acts as if the .fclist directive had been used.

Example

This example shows the assembly language and listing files for code with and without the conditional blocks listed.

Source file:

```
AAA .set   1
BB  .set   0
    .fclist
    .if    AAA
    ADD    #1024,R11
    .else
    ADD    #1024*10,R11
    .endif
    .fcnolist
    .if    AAA
    ADD    #1024,R11
    .else
    ADD    #1024*10,R11
    .endif
```

Listing file:

```
       1      0001  AAA      .set   1
       2      0000  BB       .set   0
       3                     .fclist
       4                     .if    AAA
       5 0000 503B           ADD    #1024,R11
         0002 0400
       6                     .else
       7                     ADD    #1024*10,R11
       8                     .endif
       9                     .fcnolist
      11 0004 503B           ADD    #1024,R11
         0006 0400
```

.field *Initialize Field*

Syntax

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

Description

The **.field** directive initializes a multiple-bit field within a single word (16) 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 in bits* is an optional parameter; it specifies a number from 1 to 32, which is the number of bits in the field. If you do not specify a size, the assembler assumes the size is 16. If you specify a value that cannot fit in *size in 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! line 21: W0001: Field value truncated to 1
                .field  3, 1
```

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

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

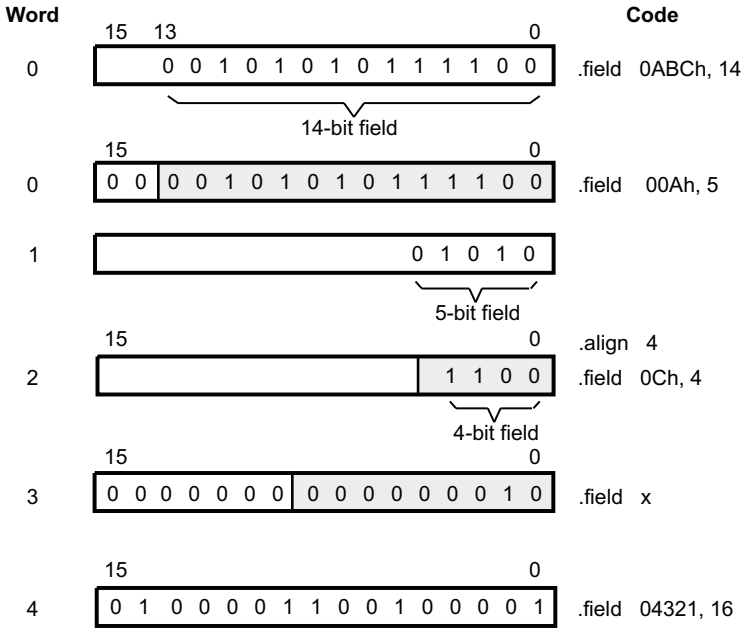
When you use .field in a .struct/.endstruct sequence, .field defines a member's size; it does not initialize memory. For more information, see the .struct/.endstruct/.tag topic .

Example

This example shows how fields are packed into a word. The SPC does not change until a word is filled and the next word is begun. Figure 4-6 shows how the directives in this example affect memory.

```
 1                ***********************************
 2                **    Initialize a 14-bit field.  **
 3                ***********************************
 4 0000 0ABC             .field  0ABCh, 14
 5
 6                ***********************************
 7                **    Initialize a 5-bit field    **
 8                **        in the same word.       **
 9                ***********************************
10 0002 000A     L_F:    .field  0Ah, 5
11
12                ***********************************
13                **       Write out the word.      **
14                ***********************************
15                       .align  4
16
17                ***********************************
18                **    Initialize a 4-bit field.   **
19                ** This fields starts a new word. **
20                ***********************************
21 0004 000C     x:      .field  0Ch, 4
22
23                ***********************************
24                **    16-bit relocatable field    **
25                **        in the next word.       **
26                ***********************************
27 0006 0004!            .field  x
28
29                ***********************************
30                **   Initialize a 16-bit field.   **
31                ***********************************
32 0008 4321             .field  04321h, 16
```

Figure 4-6. The .field Directive



.global/.def/.ref *Identify Global Symbols*

Syntax

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

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

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

Example

This example shows four files. The file1.lst and file2.lst refer to each other for all symbols used; file3.lst and file4.lst are similarly related.

The **file1.lst** and **file3.lst** files 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. Also, file1.lst uses the .global directive to identify these global symbols; file3.lst uses .ref and .def to identify the symbols.

The **file2.lst** and **file4.lst** files 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. Also, file2.lst uses the .global directive to identify these global symbols; file4.lst uses .ref and .def to identify the symbols.

file1.lst

```
 1                ; Global symbol defined in this file
 2                        .global INIT
 3                ; Global symbols defined in file2.lst
 4                        .global X, Y, Z
 5 0000           INIT:
 6 0000 503B              ADD     #56h, R11
   0002 0056
 7 0004 0000!             .word   X
 8                ;       .
 9                ;       .
10                ;       .
11                        .end
```

file2.lst

```
 1                ; Global symbols defined in this file
 2                        .global X, Y, Z
 3                ; Global symbol defined in file1.lst
 4                        .global INIT
 5 0001           X:      .set    1
 6 0002           Y:      .set    2
 7 0003           Z:      .set    3
 8 0000 0000!             .word   INIT
 9                ;       .
10                ;       .
11                ;       .
12                        .end
```

file3.lst

```
 1                ; Global symbol defined in this file
 2                        .def INIT
 3                ; Global symbols defined in file2.lst
 4                        .ref X, Y, Z
 5 0000           INIT:
 6 0000 503B              ADD     #56h, R11
   0002 0056
 7 0004 0000!             .word   X
 8                ;       .
 9                ;       .
10                ;       .
11                        .end
```

file4.lst

```
 1            ; Global symbols defined in this file
 2                    .def X, Y, Z
 3            ; Global symbol defined in file3.lst
 4                    .ref INIT
 5      0001  X:      .set    1
 6      0002  Y:      .set    2
 7      0003  Z:      .set    3
 8 0000 0000!         .word   INIT
 9            ;       .
10            ;       .
11            ;       .
12                    .end
```

.half/.short *Initialize 16-Bit Integers*

Syntax

.half *value*$_1$[, ... , *value*$_n$]

.short *value*$_1$[, ... , *value*$_n$]

Description

The **.half** and **.short** directives place one or more values into consecutive halfwords in the current section. A *value* can be either:

- An expression that the assembler evaluates and treats as a 16-bit signed or unsigned number
- A character string enclosed in double quotes. Each character in a string represents a separate value and is stored alone in the least significant eight bits of a 16-bit field, which is padded with 0s.

The assembler truncates values greater than 16 bits. You can use up to 100 value parameters, but the total line length cannot exceed 200 characters.

If you use a label with .half or .short, it points to the location where the assembler places the first byte.

These directives perform a word (16-bit) alignment before data is written to the section.

When you use .half or .short in a .struct/.endstruct sequence, they define a member's size; they do not initialize memory. For more information, see the .struct/.endstruct/.tag topic .

Example

In this example, .half is used to place 16-bit values (10, -1, abc, and a) into consecutive words in memory; .short is used to place 16-bit values (8, -3, def, and b) into consecutive words in memory. The label STRN has the value 100h, which is the location of the first initialized word for .short.

```
1 0000                .space  100h * 16
2 1000 000A           .half   10, -1, "abc", "a'
  1002 FFFF
  1004 0061
  1006 0062
  1008 0063
  100a 0061
3 100c 0008  STRN     .short  8, -3, "def", 'b'
  100e FFFD
  1010 0064
  1012 0065
  1014 0066
  1016 0062
```

.if/.elseif/.else/.endif *Assemble Conditional Blocks*

Syntax

.if *well-defined expression*

[**.elseif** *well-defined expression*]

[**.else**]

.endif

Description

Four directives provide conditional assembly:

The **.if** directive marks the beginning of a conditional block. The *well-defined expression* is a required parameter.

- If the expression evaluates to true (nonzero), the assembler assembles the code that follows the expression (up to a .elseif, .else, or .endif).
- If the expression evaluates to false (0), the assembler assembles code that follows a .elseif (if present), .else (if present), or .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 .endif (if no .elseif or .else is present). 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 a .else (if present) or a .endif.

The **.else** directive identifies a block of code that the assembler assembles when the .if expression and all .elseif expressions are false (0). The .else 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.

See Section 3.9.4 for information about relational operators.

Example

This example shows conditional assembly:

```
 1      0001  SYM1    .set    1
 2      0002  SYM2    .set    2
 3      0003  SYM3    .set    3
 4      0004  SYM4    .set    4
 5
 6            If_4:   .if     SYM4 = SYM2 * SYM2
 7 0000 0004          .byte   SYM4            ; Equal values
 8                    .else
 9                    .byte   SYM2 * SYM2     ; Unequal values
10                    .endif
11
12            If_5:   .if SYM1 <= 10
13 0001 000A          .byte   10              ; Equal values
14                    .else
15                    .byte   SYM1            ; Unequal values
16                    .endif
17
18            If_6:   .if     SYM3 * SYM2 != SYM4 + SYM2
19                    .byte   SYM3 * SYM2     ; Unequal value
20                    .else
21 0002 0008          .byte   SYM4 + SYM4       ; Equal values
22                    .endif
23
24            If_7:   .if     SYM1 = SYM2
25                    .byte   SYM1
26                    .elseif SYM2 + SYM3 = 5
27 0003 0005          .byte   SYM2 + SYM3
28                    .endif
```

.int/.word *Initialize 16-Bit Integers*

Syntax

.int *value*$_1$[**,** ... **,** *value*$_n$]

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

Description

The **.int** and **.word** directives place one or more values into consecutive words in the current section. Each value is placed in a 16-bit word by itself and is aligned on a word boundary. A *value* can be either:

- An expression that the assembler evaluates and treats as a 16-bit signed or unsigned number
- A character string enclosed in double quotes. Each character in a string represents a separate value and is stored alone in the least significant eight bits of a 16-bit field, which is padded with 0s.

A value can be either an absolute or a relocatable expression. 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 (200 characters). If you use a label with these directives, it points to the first word that is initialized.

When you use .int and .word directives in a .struct/.endstruct sequence, they define a member's size; they do not initialize memory. See the .struct/.endstruct/.tag topic .

Example

In this example, the .int and .word directives are used to initialize words. The symbol WORDX points to the first word that is reserved by .word.

```
1 0000              .space 73h
2 0000              .bss   PAGE, 128
3 0080              .bss   SYMPTR, 4
4 0074 403B  INST:  MOV    #056h, R11
  0076 0056
5 0078 000A         .int   10, SYMPTR, -1, 35 + 'a', INST, "abc"
  007a 0080!
  007c FFFF
  007e 0084
  0080 0074!
  0082 0061
  0084 0062
  0086 0063
6 0000 0C80  WORDX: .word  3200, 1 + 'AB', -0AFh, 'X'
  0002 4242
  0004 FF51
  0006 0058
```

.label *Create a Load-Time Address Label*

Syntax

.label *symbol*

Description

The **.label** directive defines a special *symbol* that refers to the load-time address rather than the run-time address within the current section. Most sections created by the assembler have relocatable addresses. The assembler assembles each section as if it started at 0, and the linker relocates it to the address at which it loads and runs.

For some applications, it is desirable to have a section load at one address and run at a *different* address. For example, you may want to load a block of performance-critical code into slower memory to save space and then move the code to high-speed memory to run it. Such a section is assigned two addresses at link time: a load address and a run address. All labels defined in the section are relocated to refer to the run-time 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 *load-time* address. This function is useful primarily to designate where the section was loaded for purposes of the code that relocates the section.

Example

This example shows the use of a load-time address label.

```
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
```

See Section 7.9 for more information about assigning run-time and load-time addresses in the linker.

.length/.width *Set Listing Page Size*

Syntax

.length [*page length*]

.width [*page width*]

Description

Two directives allow you to control the size of the output listing file.

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

- Default length: 60 lines. If you do not use the .length directive or if you use the .length directive without specifying the *page length*, the output listing length defaults to 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 the lines following. You can reset the page width with another .width directive.

- Default width: 132 characters. If you do not use the .width directive or if you use the .width directive without specifying a *page width*, the output listing width defaults to 132 characters.
- Minimum width: 80 characters
- Maximum width: 200 characters

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
```

.list/.nolist *Start/Stop Source Listing*

Syntax

.list

.nolist

Description

Two directives enable you to control the printing of the source listing:

The **.list** directive allows the printing of the source listing.

The **.nolist** directive suppresses the source listing output until a .list directive is encountered. The .nolist directive can be used to reduce assembly time and the source listing size. It can be used in macro definitions to suppress 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.

By default, the source listing is printed to the listing file; the assembler acts as if the .list directive had been used. However, if you do not request a listing file when you invoke the assembler by including the --asm_listing option on the command line (see Section 3.3), the assembler ignores the .list directive.

Example

This example shows how the .copy directive inserts 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 the .list directives do not appear in the listing file. Also, the line counter is incremented, even when source statements are not listed.

Source file:

```
   .copy "copy2.asm"
* Back in original file
   NOP
   .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
       3 0004 4303           NOP
       7              * Back in original file
       8 000a 0044           .string "Done"
         000b 006F
         000c 006E
         000d 0065
```

.long *Initialize 32-Bit Integer*

Syntax

.long *value*$_1$[**,** ... **,** *value*$_n$]

Description

The .long directive places one or more values into consecutive words in the current section. A value can be either:

- An expression that the assembler evaluates and treats as a 32-bit signed or unsigned number
- A character string enclosed in double quotes. Each character in a string represents a separate value and is stored alone in the least significant eight bits of a 32-bit field, which is padded with 0s.

A value can be either an absolute or a relocatable expression. 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.

The .long directive performs a word (16-bit) alignment before any data is written to the section.

When you use .long directive in a .struct/.endstruct sequence, it defines a member's size; it does not initialize memory. See the .struct/.endstruct/.tag topic .

Example This example shows how the .long directive initializes words. The symbol DAT1 points to the first word that is reserved.

```
1 0000 ABCD  DAT1:  .long   0ABCDh, 'A' + 100h, 'g', 'o'
  0002 0000
  0004 0141
  0006 0000
  0008 0067
  000a 0000
  000c 006F
  000e 0000
2 0010 0000!        .long   DAT1, 0AABBCCDDh
  0012 0000
  0014 CCDD
  0016 AABB
3 0018       DAT2:
```

.loop/.endloop/.break *Assemble Code Block Repeatedly*

Syntax

.loop [*well-defined expression*]

.break [*well-defined expression*]

.endloop

Description

Three directives allow 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 (the number of loops to be performed). If there is no *well-defined expression*, the loop count defaults to 1024, unless the assembler first encounters a .break directive with an expression that is true (nonzero) or omitted.

The **.break** directive, along with its expression, is optional. This means that when you use the .loop construct, you do not have to use the .break construct. The .break directive terminates a repeatable block of code only if the *well-defined expression* is true (nonzero) or omitted, and the assembler breaks the loop and assembles the code after the .endloop directive. If the expression is false (evaluates to 0), the loop continues.

The **.endloop** directive terminates a repeatable block of code; it executes when the .break directive is true (nonzero) or when the number of loops performed equals the loop count given by .loop.

Example

This example illustrates how these directives can be used with the .eval directive. The code in the first six lines expands to the code immediately following those six lines.

```
       1                    .eval     0,x
       2               COEF .loop
       3                    .word     x*100
       4                    .eval     x+1, x
       5                    .break    x = 6
       6                    .endloop
1        0000 0000          .word     0*100
1                           .eval     0+1, x
1                           .break    1 = 6
1        0002 0064          .word     1*100
1                           .eval     1+1, x
1                           .break    2 = 6
1        0004 00C8          .word     2*100
1                           .eval     2+1, x
1                           .break    3 = 6
1        0006 012C          .word     3*100
1                           .eval     3+1, x
1                           .break    4 = 6
1        0008 0190          .word     4*100
1                           .eval     4+1, x
1                           .break    5 = 6
1        000a 01F4          .word     5*100
1                           .eval     5+1, x
1                           .break    6 = 6
```

.macro/.endm *Define Macro*

Syntax

macname **.macro** [*parameter*$_1$[, ... *parameter*$_n$]]

model statements or macro directives

.endm

Description

The **.macro** and **.endm** directives are used to define macros.

You can define a macro anywhere in your program, but you must define the macro before you can use it. Macros can be defined at the beginning of a source file, in an .include/.copy file, or in a macro library.

macname	names the macro. You must place the name in the source statement's label field.
.macro	identifies the source statement as the first line of a macro definition. You must place .macro in the opcode field.
[*parameters*]	are optional substitution symbols that appear as operands for the .macro directive.
model statements	are instructions or assembler directives that are executed each time the macro is called.
macro directives	are used to control macro expansion.
.endm	marks the end of the macro definition.

Macros are explained in further detail in Chapter 5.

.mlib *Define Macro Library*

Syntax

.mlib ["]*filename*["]

Description

The **.mlib** directive provides the assembler with the *filename* of a macro library. A macro library is a collection of files that contain macro definitions. The macro definition files are bound into a single file (called a library or archive) by the archiver.

Each file in a macro library contains one macro definition that corresponds to the name of the file. 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 can be enclosed in double quotes. You can specify a full pathname (for example, c:\320tools\macs.lib). If you do not specify a full pathname, the assembler searches for the file in the following locations in the order given:

1. The directory that contains the current source file
2. Any directories named with the --include_path assembler option
3. Any directories specified by the MSP430_A_DIR environment variable
4. Any directories specified by the MSP430_C_DIR environment variable

See Section 3.4 for more information about the --include_path option.

When the assembler encounters a .mlib directive, it opens the library specified by the *filename* 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 way it expands 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.

See Chapter 5 for more information on macros and macro libraries.

Example

This example creates a macro library that defines two macros, inc4 and dec4. The file inc4.asm contains the definition of inc4, and dec4.asm contains the definition of dec4.

inc4.asm

```
* Macro for incrementing
inc4  .macro reg
     ADD.W #1, reg
     ADD.W #1, reg
     ADD.W #1, reg
     ADD.W #1, reg
     .endm
```

dec4.asm

```
* Macro for decrementing
dec4  .macro reg
     SUB.W #1, reg
     SUB.W #1, reg
     SUB.W #1, reg
     SUB.W #1, reg
     .endm
```

Use the archiver to create a macro library:

```
ar430 -a mac inc4.asm dec4.asm
```

Now you can use the .mlib directive to reference the macro library and define the inc4 and dec4 macros:

```
       1                    .mlist
       2                    .mlib "mac.lib"
       3              ; Macro call
       4 0000               inc4  R11, R12, R13, R14
1        0000 531B            ADD.W  #1,R11
1        0002 531C            ADD.W  #1,R12
1        0004 531D            ADD.W  #1,R13
1        0006 531E            ADD.W  #1,R14
       5
       6              ; Macro call
       7 0008               dec4  R11, R12, R13, R14
1        0008 831B            SUB.W  #1,R11
1        000a 831C            SUB.W  #1,R12
1        000c 831D            SUB.W  #1,R13
1        000e 831E            SUB.W #1,R14
```

.mlist/.mnolist *Start/Stop Macro Expansion Listing*

Syntax

.mlist

.mnolist

Description

Two directives enable you 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 suppresses macro and .loop/.endloop block expansions in the listing file.

By default, the assembler behaves as if the .mlist directive had been specified.

See Chapter 5 for more information on macros and macro libraries. See the .loop/.break/.endloop topic for information on conditional blocks.

Example

This example defines 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   P1, P2, P3
       2                     .string ":p1:", ":p2:", ":p3:"
       3                     .endm
       4
       5 0000                STR_3 "as", "I", "am"   ; Invoke STR_3 macro.
1        0000 003A           .string ":p1:", ":p2:", ":p3:"
         0001 0070
         0002 0031
         0003 003A
         0004 003A
         0005 0070
         0006 0032
         0007 003A
         0008 003A
         0009 0070
         000a 0033
         000b 003A
       6                     .mnolist                ; Suppress expansion.
       7 000c                STR_3 "as", "I", "am"   ; Invoke STR_3 macro.
       8                     .mlist                  ; Show macro expansion.
       9 0018                STR_3 "as", "I", "am"   ; Invoke STR_3 macro.
1        0018 003A           .string ":p1:", ":p2:", ":p3:"
         0019 0070
         001a 0031
         001b 003A
         001c 003A
         001d 0070
         001e 0032
         001f 003A
         0020 003A
         0021 0070
         0022 0033
         0023 003A
```

.newblock *Terminate Local Symbol Block*

Syntax

.newblock

Description

The **.newblock** directive undefines any local labels currently defined. Local labels, by nature, are temporary; the .newblock directive resets them and terminates their scope.

A local label is a label in the form $*n*, where *n* is a single decimal digit, or *name*?, where *name* is a legal symbol name. Unlike other labels, local labels are intended to be used locally, cannot be used in expressions, and do not qualify for branch expansion if used with a branch. They can be used only as operands in 8-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. The .text, .data, and .sect directives also reset local labels. Local labels that are defined within an include file are not valid outside of the include file.

See Section 3.8.2 for more information on the use of local labels.

Example

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

```
 1                      .global ADDRA,ADDRB,ADDRC
 2
 3 0000 403B  Label1: MOV #ADDRA, R11  ; Load Address A to R11
   0002 0000!
 4 0004 803B          SUB #ADDRB, R11  ; Subtract Address B.
   0006 0000!
 5 0008 3803          JL $1            ; If &lt; 0, branch to $1
 6 000a 403B          MOV #ADDRB, R11  ; otherwise, load ADDRB to R11
   000c 0000!
 7 000e 3C02          JMP $2           ; and branch to $2
 8 0010 403B  $1      MOV #ADDRA, R11  ; $1: load ADDRA to AC0.
   0012 0000!
 9 0014 503B  $2      ADD #ADDRC, R11  ; $2: add ADDRC.
   0016 0000!
10                    .newblock        ; Undefine $1 so can be used again.
11 0018 3C02          JMP $1           ; If less than zero,branch to $1.
12 001a 4B82          MOV R11,&ADDRC   ; Store AC0 low in ADDRC.
   001c 0000!
13 001e 4303  $1      NOP
```

.option *Select Listing Options*

Syntax

.option *option*$_1$**[**, *option*$_2$**,** . . .]

Description

The **.option** directive selects options for the assembler output listing. The *options* must be separated by commas; each option selects a listing feature. These are valid options:

A	turns on listing of all directives and data, and subsequent expansions, macros, and blocks.
B	limits the listing of .byte and .char directives to one line.
H	limits the listing of .half and .short directives to one line.
M	turns off macro expansions in the listing.
N	turns off listing (performs .nolist).
O	turns on listing (performs .list).
R	resets the B, H, M, T, and W directives (turns off the limits of B, H, M, T, and W).
T	limits the listing of .string directives to one line.
W	limits the listing of .word and .int 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 --cross_reference option (see Section 3.3).

Options *are not* case sensitive.

Example

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

```
                ************************************************
                ** Limit the listing of .byte, .char, .int, .long, **
                **  .word, and .string directives to 1 line each.  **
                ************************************************
                        .option B, W, T
0000 00BD               .byte   -'C', 0B0h, 5
0003 00BC               .char   -'D', 0C0h, 6
0006 000A               .int    10, 35 + 'a', "abc"
0010 CCDD               .long   0AABBCCDDh, 536 + 'A'
0012 AABB
0014 0259
0016 0000
0018 15AA               .word   5546, 78h
001c 0045               .string "Extended Registers"

                ************************************************
                **          Reset the listing options.          **
                ************************************************
                        .option R
002e 00BD               .byte   -'C', 0B0h, 5
002f 00B0
0030 0005
0031 00BC               .char   -'D', 0C0h, 6
0032 00C0
0033 0006
0034 000A               .int    10, 35 + 'a', "abc"
0036 0084
0038 0061
003a 0062
003c 0063
003e CCDD               .long   0AABBCCDDh, 536 + 'A'
0040 AABB
0042 0259
0044 0000
0046 15AA               .word   5546, 78h
0048 0078
```

```
                    22 004a 0045          .string "Extended Registers"
                       004b 0078
                       004c 0074
                       004d 0065
                       004e 006E
                       004f 0064
                       0050 0065
                       0051 0064
                       0052 0020
                       0053 0052
                       0054 0065
                       0055 0067
                       0056 0069
                       0057 0073
                       0058 0074
                       0059 0065
                       005a 0072
                       005b 0073
```

.page *Eject Page in Listing*

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 assembler increments the line counter when it encounters the .page directive. 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:

```
MSP430 COFF Assembler PC vx.x.x Mon Jul 26 10:58:19 2004

Tools Copyright (c) 2003-2004 Texas Instruments Incorporated
**** Page Directive Example ****                                    PAGE 1

       2            ;         .
       3            ;         .
       4            ;         .
MSP430 COFF Assembler PC vx.x.x Mon Jul 26 10:58:19 2004

Tools Copyright (c) 2003-2004 Texas Instruments Incorporated
**** Page Directive Example ****                                    PAGE 2
```

.sect *Assemble Into Named Section*

Syntax

.sect "*section name***"**

Description

The **.sect** directive defines a named section that can be used like the default .text and .data sections. The .sect directive tells the assembler to begin assembling source code into the named section.

The *section name* identifies the section. The section name is significant to 200 characters and must be enclosed in double quotes. A section name can contain a subsection name in the form *section name*:*subsection name*.

See Chapter 2 for more information about sections.

Example

This example defines a special-purpose section, Vars, and assembles code into it.

```
 1              ************************************************
 2              **     Begin assembling into .text section.      **
 3              ************************************************
 4 0000              .text
 5 0000 403B         MOV #0x78,R11
   0002 0078
 6 0004 503B         ADD #0x78,R11
   0006 0078
 7              ************************************************
 8              **     Begin assembling into Vars section.       **
 9              ************************************************
10 0000              .sect Vars
11 0000 CCCD         .float 0.05
   0002 3D4C
12 0004 00AA  X:     .word 0xAA
13             ************************************************
14             **     Resume assembling into .text section.     **
15             ************************************************
16 0008              .text
17 0008 5B0C         ADD R11,R12
18             ************************************************
19             **     Resume assembling into Vars section.      **
20             ************************************************
21 0006              .sect Vars
22 0006 000D         .field 13
23 0008 000A         .field 0xA
24 000a 0010         .field 0x10
```

.set/.equ *Define Assembly-Time Constant*

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* is a label that 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.

Symbols defined with .set or .equ can be made externally visible with the .def or .global directive (see the .global/.def/.ref topic). In this way, you can define global absolute constants.

Example

This example shows how symbols can be assigned with .set and .equ.

```
 1              ****************************************************
 2              **   Equate symbol ACCUM to register R11 and use   **
 3              **            it instead of the register.         **
 4              ****************************************************
 5 000B         ACCUM   .set R11
 6 0000 401B            MOV 0x56, ACCUM
   0002 0054
 7
 8              ****************************************************
 9              **    Set symbol INDEX to an integer expression    **
10              **       and use it as an immediate operand.       **
11              ****************************************************
12 0035         INDEX   .equ 100/2 + 3
13 0004 503B            ADD #INDEX, ACCUM
   0006 0035
14
15              ****************************************************
16              **  Set symbol SYMTAB to a relocatable expression  **
17              **       and use it as a relocatable operand.      **
18              ****************************************************
19 0008 000A   LABEL   .word  10
20       0009! SYMTAB  .set   LABEL + 1
21
22              ****************************************************
23              **    Set symbol NSYMS equal to the symbol INDEX   **
24              **          and use it as you would INDEX.         **
25              ****************************************************
26       0035  NSYMS   .set   INDEX
27 000a 0035           .word  NSYMS
```

.space/.bes *Reserve Space*

Syntax

[*label*] **.space** *size in bytes*

[*label*] **.bes** *size in bytes*

Description

The **.space** and **.bes** directives reserve the number of bytes given by *size in bytes* in the current section and fill them with 0s. The section program counter is incremented to point to the word following the reserved space.

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

Example

This example shows how memory is reserved with the .space and .bes directives.

```
 1                ************************************************
 2                **   Begin assembling into the .text section.   **
 3                ************************************************
 4 0000                  .text
 5
 6                ************************************************
 7                **   Reserve 0F0 bytes in the .text section.    **
 8                ************************************************
 9 0000                  .space  0F0h
10 00f0 0100             .word   100h, 200h
   00f2 0200
11                ************************************************
12                **   Begin assembling into the .data section.   **
13                ************************************************
14 0000                  .data
15 0000 0049             .string "In .data"
   0001 006E
   0002 0020
   0003 002E
   0004 0064
   0005 0061
   0006 0074
   0007 0061
16                ************************************************
17                ** Reserve 100 bytes in the .data section; RES_1 **
18                **    points to the first byte that contains     **
19                **                reserved bytes.                **
20                ************************************************
21 0008           RES_1:  .space  100
22 006c 000F             .word   15
23 006e 0008!            .word   RES_1
24                ************************************************
25                **  Reserve 20 bits in the .data section; RES_2  **
26                **     points to the last byte that contains     **
27                **                 reserved bytes.               **
28                ************************************************
29 0083           RES_2:  .bes    20
30 0084 0036             .word   36h
31 0086 0083!            .word   RES_2
```

.sslist/.ssnolist *Control Listing of Substitution Symbols*

Syntax

.sslist

.ssnolist

Description

Two directives allow 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 suppresses substitution symbol expansion in the listing file.

By default, all substitution symbol expansion in the listing file is suppressed; the assembler acts as if the .ssnolist directive had been used.

Lines with the pound (#) character denote expanded substitution symbols.

Example

This example shows code that, by default, suppresses the listing of substitution symbol expansion, and it shows the .sslist directive assembled, instructing the assembler to list substitution symbol code expansion.

```
       1           SHIFT  .macro  dst,amount
       2                  .loop   amount
       3                  RLA     dst
       4                  .endloop
       5                  .endm
       6
       7                  .global value
       8
       9 0000             SHIFT   R5,3
1                         .loop   3
1                         RLA     dst
1                         .endloop
2         0000 5505       RLA     R5
2         0002 5505       RLA     R5
2         0004 5505       RLA     R5
      10 0006 5582        ADD     R5,&value
         0008 0000!
      11
      12                  .sslist
      13
      14 000a             SHIFT   R5,3
1                         .loop   amount
#                         .loop   3
1                         RLA     dst
1                         .endloop
2         000a 5505       RLA     dst
#                         RLA     R5
2         000c 5505       RLA     dst
#                         RLA     R5
2         000e 5505       RLA     dst
#                         RLA     R5
```

.string *Initialize Text*

Syntax

.string *{expr$_1$ | "string$_1$"}[, ... , {expr$_n$ | "string$_n$"}]*

Description

The **.string** directive places 8-bit characters from a character string into the current section. The *expr* or *string* can be one of the following:

- An expression that the assembler evaluates and treats as an 8-bit signed number.
- A character string enclosed in double quotes. Each character in a string represents a separate value, and values are stored in consecutive bytes. The entire string *must* be enclosed in quotes.

The assembler truncates any values that are greater than eight bits. You can have up to 100 operands, but they must fit on a single source statement line.

If you use a label, it points to the location of the first byte that is initialized.

When you use .string in a .struct/.endstruct sequence, .string defines a member's size; it does not initialize memory. For more information, see the .struct/.endstruct/.tag topic .

Example

This example shows 8-bit values placed into words in the current section.

```
1 0000 0041  Str_Ptr:    .string  "ABCD"
  0001 0042
  0002 0043
  0003 0044
2 0004 0041              .string  41h, 42h, 43h, 44h
  0005 0042
  0006 0043
  0007 0044
3 0008 0041              .string  "Austin", "Houston", "Dallas"
  0009 0075
  000a 0073
  000b 0074
  000c 0069
  000d 006E
  000e 0048
  000f 006F
  0010 0075
  0011 0073
  0012 0074
  0013 006F
  0014 006E
  0015 0044
  0016 0061
  0017 006C
  0018 006C
  0019 0061
  001a 0073
4 001b 0030              .string  36 + 12
```

.struct/.endstruct/.tag *Declare Structure Type*

Syntax

[*stag*] **.struct** [*expr*]
[mem_0] *element* [$expr_0$]
[mem_1] *element* [$expr_1$]
. . .
. . .
. . .
[mem_n] **.tag** *stag* [$expr_n$]
. . .
. . .
. . .
[mem_N] *element* [$expr_N$]
[*size*] **.endstruct**
label **.tag** *stag*

Description

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

The **.endstruct** directive terminates the structure definition.

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

Following are descriptions of the parameters used with the .struct, .endstruct, and .tag directives:

- The *stag* is the structure's tag. Its value is associated with the beginning of the structure. If no stag is present, the assembler puts the structure members in the global symbol table with the value of their absolute offset from the top of the structure. A .stag is optional for .struct, but is required for .tag.
- The *expr* is an optional expression indicating the beginning offset of the structure. The default starting point for a structure is 0.
- The $mem_{n/N}$ is an optional label for a member of the structure. This label is absolute and equates to the present offset from the beginning of the structure. A label for a structure member cannot be declared global.
- The *element* is one of the following descriptors: .byte, .char, .word, .int, .long, .string, .double, .float, .half, .short, 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 stag must be used (as in the definition of stag).
- The $expr_{n/N}$ is an optional expression for the number of elements described. This value defaults to 1. A .string element is considered to be one byte in size, and a .field element is one bit.
- The *size* is an optional label for the total size of the structure.

Directives That Can Appear in a .struct/.endstruct Sequence

Note: The only directives that can appear in a .struct/.endstruct sequence are element descriptors, conditional assembly directives, and the .align directive, which aligns the member offsets on word boundaries. Empty structures are illegal.

The following examples show various uses of the .struct, .tag, and .endstruct directives.

Example 1

```
 1                REAL_REC    .struct                 ;stag
 2          0000  NOM         .int                    ;member1 = 0
 3          0002  DEN         .int                    ;member2 = 2
 4          0004  REAL_LEN    .endstruct              ;real_len = 4
 5
 6 0000                       .bss    REAL, REAL_LEN
 7
 8 0000                       .text
 9 0000 521B                  ADD.W   &REAL + REAL_REC.DEN,R11
   0002 0002!
10
```

Example 2

```
11 0000                       .data
12                CPLX_REC    .struct
13          0000  REALI       .tag    REAL_REC        ; stag
14          0004  IMAGI       .tag    REAL_REC        ; member1 = 0
15          0008  CPLX_LEN    .endstruct              ; cplx_len = 8
16
17                COMPLEX     .tag    CPLX_REC        ; assign structure attrib
18
19 0004                       .bss    COMPLEX, CPLX_LEN
20
21 0004                       .text
22 0004 521B                  ADD   &COMPLEX.REALI,R11 ; access structure
   0006 0004!
```

Example 3

```
 1 0000                       .data
 2                            .struct                 ; no stag puts mems into
 3                                                    ; global symbol table
 4          0000  X           .int                    ; create 3 dim templates
 5          0002  Y           .int
 6          0004  Z           .int
 7          0006              .endstruct
```

Example 4

```
 1 0000                       .data
 2                 BIT_REC    .struct                 ; stag
 3          0000   STREAM     .string 64
 4          0040   BIT7       .field  7               ; bits1 = 64
 5          0040   BIT9       .field  9               ; bits2 = 64
 6          0042   BIT10      .field  10              ; bits3 = 65
 7          0044   X_INT      .int                    ; x_int = 66
 8          0046   BIT_LEN    .endstruct              ; length = 67
 9
10                 BITS       .tag    BIT_REC
11
12 0000                       .bss    BITS, BIT_REC
13
14 0000                       .text
15 0000 521B                  ADD     &BITS.BIT7,R11  ; move into R11
   0002 0040!
16 0004 F03B                  AND     #127,R11        ; mask off garbage bits
   0006 007F
```

.tab *Define Tab Size*

Syntax

.tab *size*

Description

The **.tab** directive defines the tab size. Tabs encountered in the source input are translated to *size* character spaces in the listing. The default tab size is eight spaces.

Example

In this example, each of the lines of code following a .tab statement consists of a single tab character followed by an NOP instruction.

Source file:

```
; default tab size
    NOP
    NOP
    NOP
      .tab 4
    NOP
    NOP
    NOP
      .tab 16
    NOP
    NOP
    NOP
```

Listing file:

```
      1            ; default tab size
      2 0000 4303          NOP
      3 0002 4303          NOP
      4 0004 4303          NOP
      5
      7 0006 4303      NOP
      8 0008 4303      NOP
      9 000a 4303      NOP
     10
     12 000c 4303                  NOP
     13 000e 4303                  NOP
     14 0010 4303                  NOP
```

.text *Assemble Into the .text Section*

Syntax

.text

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 .text section is the default section. Therefore, at the beginning of an assembly, the assembler assembles code into the .text section unless you use a .data or .sect directive to specify a different section.

For more information about sections, see Chapter 2.

Example

This example assembles code into the .text and .data sections.

```
 1              ****************************************
 2              * Begin assembling into the .data section. *
 3              ****************************************
 4 0000                 .data
 5 0000 000A            .byte 0Ah, 0Bh
   0001 000B
 6 0002 0011  coeff     .word 011h,0x22,0x33
   0004 0022
   0006 0033
 7
 8              ****************************************
 9              * Begin assembling into the .text section. *
10              ****************************************
11 0000                 .text
12 0000 0041  START:    .string "A", "B", "C"
   0001 0042
   0002 0043
13 0003 0058  $END:     .string "X", "Y", "Z"
   0004 0059
   0005 005A
14 0006 403A            MOV.W   #0x1234,R10
   0008 1234
15 000a 521A            ADD.W   &coeff+1,R10
   000c 0003!
16
17              ****************************************
18              *   Resume assembling into .data section.  *
19              ****************************************
20 0008                 .data
21 0008 000C            .byte   0Ch, 0Dh
   0009 000D
22
23              ****************************************
24              *   Resume assembling into .text section.  *
25              ****************************************
26 000e                 .text
27 000e 0051            .string "QUIT"
   000f 0055
   0010 0049
   0011 0054
```

.title *Define Page Title*

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 64 characters. If you supply more than 64 characters, the assembler truncates the string and issues a warning:

```
*** WARNING! line x: W0001: String is too long - will be truncated
```

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, the first source statement must contain a .title directive.

Example

In this example, one title is printed on the first page and a different title is printed on succeeding pages.

Source file:

```
       .title  "**** Fast Fourier Transforms ****"
;        .
;        .
;        .
       .title  "**** Floating-Point Routines ****"
       .page
```

Listing file:

```
MSP430 COFF Assembler PC vx.x.x Mon Jul 26 12:43:54 2004

Tools Copyright (c) 2003-2004 Texas Instruments Incorporated
**** Fast Fourier Transforms ****                                    PAGE 1

       2            ;        .
       3            ;        .
       4            ;        .
MSP430 COFF Assembler PC vx.x.x Mon Jul 26 12:43:54 2004

Tools Copyright (c) 2003-2004 Texas Instruments Incorporated
**** Floating-Point Routines ****                                    PAGE 2
```

.usect *Reserve Uninitialized Space*

Syntax

symbol **.usect "***section name***",** *size in bytes* [**,** *alignment*]

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 that space has no contents. However, .usect defines additional sections 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 for which you are reserving space.
- The *section name* is significant to 200 characters and must be enclosed in double quotes. This parameter names the uninitialized section. A section name can contain a subsection name in the form *section name*:*subsection name*.
- The *size in bytes* is an expression that defines the number of bytes that are reserved in *section name*.
- The *alignment* is an optional parameter that ensures that the space allocated to the symbol occurs on the specified boundary. The boundary indicates the size of the alignment in bytes and can be set to a power of 2 between 2^0 and 2^{15}, inclusive. If the SPC is aligned at the specified boundary, it is not incremented.

Initialized sections directives (.text, .data, and .sect) end the current section and tell the assembler to begin assembling into another section. A .usect or .bss directive encountered in the current section is simply assembled, and assembly continues in the current section.

Variables that can be located contiguously in memory can be defined in the same specified section; to do so, repeat the .usect directive with the same section name and the subsequent symbol (variable name).

For more information about sections, see Chapter 2.

Example

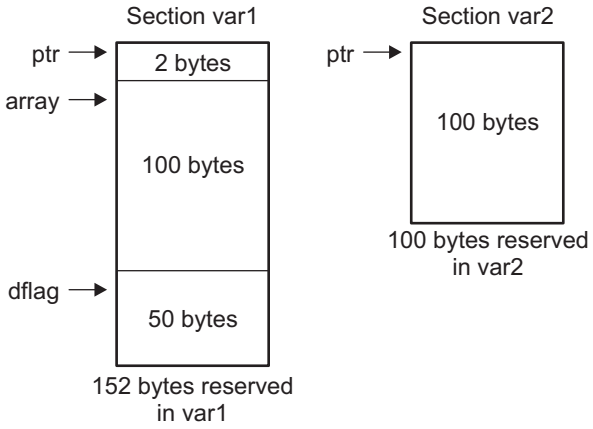
This example uses the .usect directive to define two uninitialized, named sections, var1 and var2. The symbol ptr points to the first byte reserved in the var1 section. The symbol array points to the first byte in a block of 100 bytes reserved in var1, and dflag points to the first byte in a block of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

Figure 4-7 shows how this example reserves space in two uninitialized sections, var1 and var2.

```
 1                ****************************************************
 2                **         Assemble into the .text section.       **
 3                ****************************************************
 4 0000                    .text
 5 0000 403B               MOV #0x3,R11
   0002 0003
 6
 7                ****************************************************
 8                **       Reserve 2 bytes in the var1 section.     **
 9                ****************************************************
10 0000           ptr      .usect "var1",2
11
12                ****************************************************
13                **      Reserve 100 bytes in the var1 section.    **
14                ****************************************************
15 0002           array    .usect "var1",100
16
17 0004 503B              ADD #37,R11
   0006 0025
18
19                ****************************************************
20                **       Reserve 50 bytes in the var1 section.    **
21                ****************************************************
22 0066           dflag    .usect "var1",50
23
```

```
24 0008 503C          ADD #dflag-array,R12
   000a 0064
25
26             ****************************************************
27             ** Reserve 100 bytes in the var2 section. **
28             ****************************************************
29 0000        vec     .usect "var2",100
30
31 000c 5C0B          ADD R12,R11
32
33             ****************************************************
34             **     Declare a .usect symbol to be external.      **
35             ****************************************************
36                     .global array
```

Figure 4-7. The .usect Directive



.var *Use Substitution Symbols as Local Variables*

Syntax

.var sym_1[, sym_2, ... , sym_n]

Description

The .var directive allows you to use substitution symbols as local variables within a macro. With this directive, you can define up to 32 local macro substitution symbols (including parameters) per macro.

The .var directive creates temporary substitution symbols with the initial value of the null string. These symbols are not passed in as parameters, and they are lost after expansion.

See Chapter 5 for information on macros.