

Topics

3	Assembler Description	3-3
3.1	Assembler Development Flow	3-4
3.2	Invoking the Assembler	3-5
3.3	Naming Alternate Directories for Assembler Input	3-7
3.3.1	-i Assembler Option	3-7
3.3.2	Environment Variable (A_DIR)	3-8
3.4	Source Statement Format	3-9
3.5	Constants	3-12
3.6	Character Strings	3-14
3.7	Symbols	3-15
3.8	Expressions	3-18
3.8.1	Operators	3-19
3.8.2	Expression Overflow and Underflow	3-19
3.8.3	Well-Defined Expressions	3-20
3.8.4	Conditional Expressions	3-20
3.8.5	Relocatable Symbols and Legal Expressions	3-20
3.9	Source Listings	3-23
3.10	Cross-Reference Listings	3-26

Examples

Ex.	Title	Page
3.1	An Assembler Listing	3-25
3.2	An Assembler Cross-Reference Listing	3-26

Figures

Fig.	Title	Page
3.1	Assembler Development Flow	3-4

Tables

Table	Title	Page
3.1	Operators Used in Expressions (Precedence)	3-19
3.2	Expressions With Absolute and Relocatable Symbols	3-21
3.3	Symbol Attributes	3-26

3 Assembler Description

The assembler translates assembly language source files into machine language object files. These files are in common object file format (COFF). Source files can contain the following assembly language elements:

Assembler directives

Assembly language instructions

Macro directives

This two-pass assembler does the following:

- Processes the source statements in a text file to produce a relocatable object file.
- Produces a source listing (if requested) and provides you with control over this listing.
- Allows you to segment your code into sections and maintain an SPC (section program counter) for each section of object code.
- Defines and references global symbols and appends a cross-reference listing to the source listing (if requested).
- Assembles conditional blocks.
- Supports macros, allowing you to define macros inline or in a library.

3.1 Assembler Development Flow

The figure illustrates the assembler's role in the assembly language development flow. The assembler accepts assembly language source files as input.

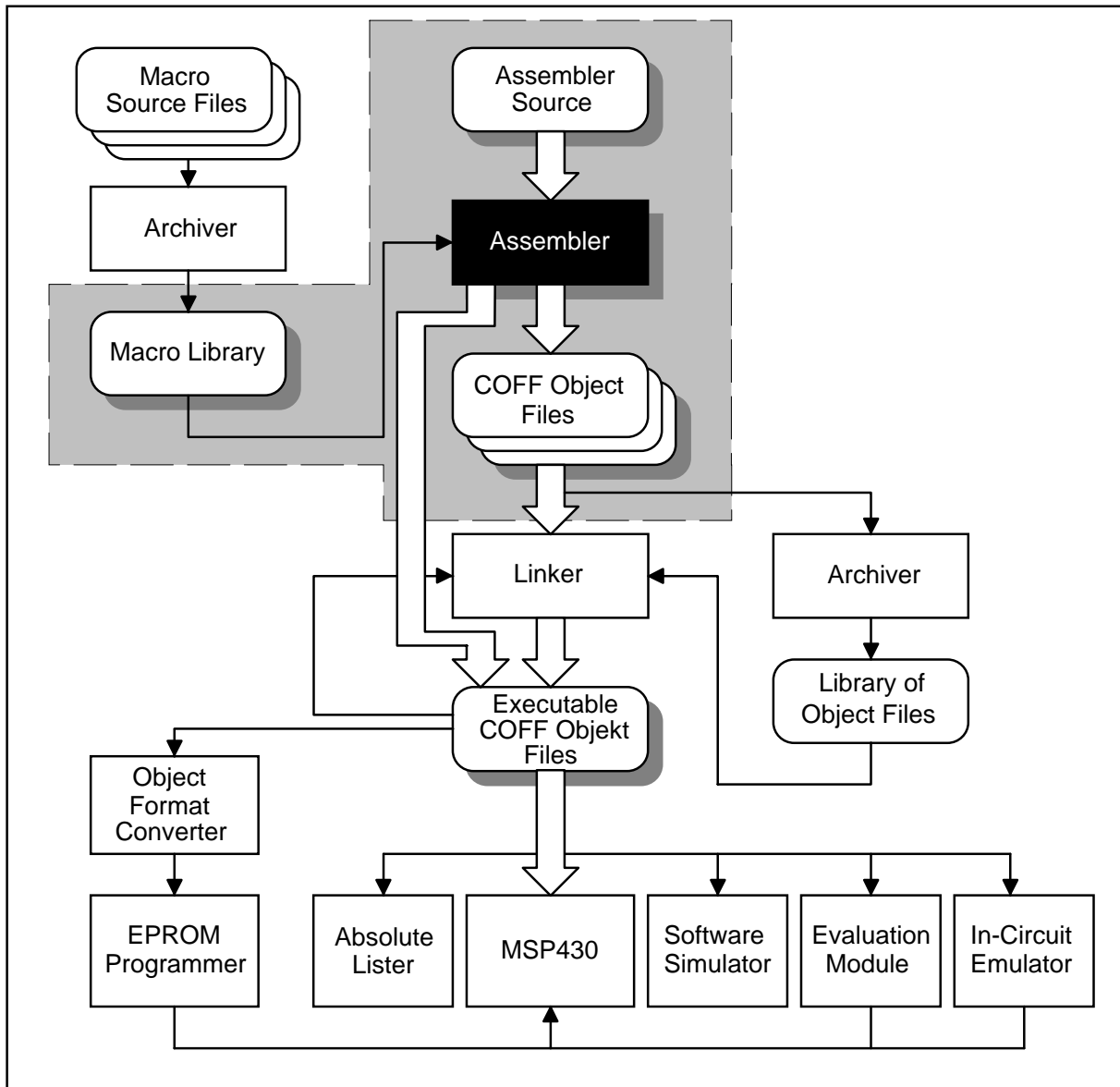


Figure 3.1: Assembler Development Flow

3.2 Invoking the Assembler

To invoke the assembler, enter the following:

asm430 <i>[input file [object file [listing file [rawdata file]]]] [-options]</i>

asm430 is the command that invokes the assembler.

input file names the assembly language source file. If you do not supply an extension, the assembler assumes that the input file has the default extension *.asm*. If you do not supply an input filename when you invoke the assembler, the assembler will prompt you for one.

object file names the object file that the assembler creates. If you do not supply an extension, the assembler uses *.obj* as a default extension. If you do not supply an object file, the assembler creates a file that uses the input file name with the *.obj* extension.

listing file names the optional listing file that the assembler can create. If you do not supply a name for a listing file, *the assembler does not create one* unless you use the *-l* option. In this case, the assembler uses the input filename. If you do not supply an extension, the assembler uses *.lst* as a default extension.

rawdata file names the optional ascii-format object file that the assembler can create. If you do not supply a name for the rawdata file, the assembler does not create one unless you use the *-z* option. In this case, the assembler uses the input filename. If you do not supply an extension, the assembler uses *.txt* as a default extension.

options identifies the assembler options that you want to use.

Options *are not* case-sensitive and can appear anywhere on the command line following the command. Precede each option with a hyphen (-). You can string the options together; for example, *-lc* is equivalent to *-l -c*. The valid assembler options are as follows:

- a** creates an absolute listing. When you use *-a*, the assembler does not produce an object file. The absolute listing option is used in conjunction with the absolute lister.
- b** suppress banner on all pages except page 1.
- c** makes case insignificant. For example, the symbols *ABC* and *abc* will be equivalent. *If you do not use this option, case is significant.*
- i** specifies a directory where the assembler can find files named by the *.copy*, *.include*, or *.mlib* directives. The format of the *-i* option is *-ipath-name*. You can specify up to 10 directories in this manner; each path-name must be preceded by the *-i* option.
- l** (lowercase "L") produces a listing file.
- q** (quiet) suppresses the banner and all progress information.

- s** puts **all** defined symbols in the object file's symbol table. Usually, the assembler puts only global symbols into the symbol table. When you use -s, symbols that are defined as labels or as assembly-time constants are also placed in the symbol table.
- x** produces a cross-reference table and appends it to the end of the listing file. If you do not request a listing file, the assembler creates one anyway.
- z** creates an object file in ascii format containing no relocation and debug information. The generation of the COFF object file is not affected. This option is used in conjunction with the evaluation module.

3.3 Naming Alternate Directories for Assembler Input

The `.copy`, `.include`, and `.mlib` directives tell the assembler to use code from external files. The `.copy` and `.include` directives tell the assembler to read source statements from another file, and the `.mlib` directive names a library that contains macro functions. The syntaxes for these directives are:

```
.copy    "filename"
```

```
.include "filename"
```

```
.mlib   "filename"
```

The *filename* names a copy/include file that the assembler reads statements from or a macro library that contains macro definitions. The *filename* may be a complete pathname, a partial pathname, or a filename with no path information. The assembler searches for the file in:

- 1) The directory that contains the current source file. The current source file is the file being assembled when the `.copy`, `.include`, or `.mlib` directive is encountered.
- 2) Any directories named with the `-i` assembler option.
- 3) Any directories set with the environment variable `A_DIR`.

You can augment the assembler's directory search algorithm by using the `-i` assembler option or the `A_DIR` environment variable.

3.3.1 -i Assembler Option

The `-i` assembler option names an alternate directory that contains copy/include files or macro libraries. The format of the `-i` option is as follows:

```
asm430 -ipathname source filename
```

You can use up to 10 `-i` options per invocation; each `-i` option names one *pathname*. In assembly source, you can use the `.copy`, `.include`, or `.mlib` directive without specifying any path information. If the assembler doesn't find the file in the directory that contains the current source file, it searches the paths provided by the `-i` options.

For example, assume that a file called `source.asm` is in the current directory; `source.asm` contains the following directive statement:

```
.copy "copy.asm"
```

	Pathname for <code>copy.asm</code>	Invocation Command
DOS	<code>c:\430\files\copy.asm</code>	<code>asm430 -ic:\430\files source.asm</code>

The assembler first searches for `copy.asm` in the current directory because `source.asm` is in the current directory. Then, the assembler searches in the directory named with the `-i` option.

3.3.2 Environment Variable (`A_DIR`)

An environment variable is a system symbol that you define and assign a string to. The assembler uses the environment variable **`A_DIR`** to name alternate directories that contain copy/include files or macro libraries. The command for assigning the environment variable is as follows:

DOS `set A_DIR = pathname;another pathname ...`

The *pathnames* are directories that contain copy/include files or macro libraries. You can separate the pathnames with a semicolon or with blanks. In assembly source, you can use the `.copy`, `.include`, or `.mlib` directive without specifying any path information. If the assembler doesn't find the file in the directory that contains the current source file or in directories named by `-i`, it searches the paths named by the environment variable.

For example, assume that a file called `source.asm` contains these statements:

```
.copy "copy1.asm"
.copy "copy2.asm"
```

	Pathname	Invocation Command
DOS	<code>c:\430\files\copy1.asm</code> <code>c:\dsys\copy2.asm</code>	<code>set A_DIR=c:\dsys; c:\exec\source</code> <code>asm430 -ic:\430\files source.asm</code>

The assembler first searches for `copy1.asm` and `copy2.asm` in the current directory because `source.asm` is in the current directory. Then the assembler searches in the directory named with the `-i` option and finds `copy1.asm`. Finally, the assembler searches the directory named with `A_DIR` and finds `copy2.asm`.

Note that the environment variable remains set until you reboot the system or reset the variable by entering one of these commands:

DOS `set A_DIR=`

3.4 Source Statement Format

MSP430 assembly language source programs consist of source statements that can contain assembler directives, assembly language instructions, macro directives, and comments. Source statement lines can be as long as the source file format allows, but the assembler reads up to 200 characters per line. If a statement contains more than 200 characters, the assembler truncates the line and issues a warning.

The next several lines show examples of source statements:

```
sym      .equ      2           ; Symbol sym = 2
Begin:   ADD      #sym+5,R11   ; Add (sym+5) to contents of R11
          .word    016h        ; Initialize a word with 016h
```

A source statement can contain four ordered fields. The general syntax for source statements is as follows:

[label] [:] mnemonic [operand list] [;comment]

Follow these guidelines:

- All statements must begin with a label, a blank, an asterisk, or a semicolon.
- Labels are optional; if used, they must begin in column 1.
- One or more blanks must separate each field. Note that tab characters are equivalent to blanks.
- Comments are optional. Comments that begin in column 1 can begin with an asterisk or a semicolon (* or ;), but comments that begin in any other column **must** begin with a semicolon.

Label Field

Labels are optional for all assembly language instructions and for most (but not all) assembler directives. When used, a label **must** begin in column 1 of a source statement. A label can contain up to 32 alphanumeric characters (A-Z, a-z, 0-9, _, and \$). Labels are case-sensitive, and the first character cannot be a number. A label can be followed by a colon (:); the colon is not treated as part of the label name. If you don't use a label, the first character position must contain a blank, a semicolon, or an asterisk.

When you use a label, its value is the current value of the section program counter (the label points to the statement it's associated with). If, for example, you use the .word directive to initialize several words, a label would point to the first word. In the following example, the label Start has the value 40h.

```
.      .      .      .
.      .      .      .
.      .      .      .
9      003F          * Assume some other code was assembled
10     0040      000A  Start: .word 0Ah,3,7
      0041      0003
      0043      0007
```

A label on a line by itself is a valid statement. It assigns the current value of the section program counter to the label; this is equivalent to the following directive statement:

```
label    .equ    $ ;    $ provides the current value of the SPC
```

When a label appears on a line by itself, it points to the instruction on the next line (the SPC is not incremented):

```
3          0050          Here:
4          0050    0003          .word 3
```

Mnemonic Field

The mnemonic field follows the label field. *The mnemonic field cannot start in column 1, or it would be interpreted as a label.* The mnemonic field can contain one of the following opcodes:

- Machine-instruction mnemonic (such as ADC, MOV, POP)
- Assembler directive (such as .data, .list, .equ)
- Macro directive (such as .macro, .var, .mexit)
- A macro call

Operand Field

The operand field is a list of operands that follow the mnemonic field. An operand can be a constant, a symbol, or a combination of constants and symbols in an expression. You must separate operands with commas.

- **Operand Prefixes for Instructions**

The assembler allows you to specify that a constant, symbol, or expression should be used as an address, an immediate value, or an indirect value. The following rules apply to the operands of instructions.

- **No prefix — the operand is an address or a register.** If you do not use a prefix with an operand, the assembler treats an operand representing a constant value as an absolute address. When the operand is a label, the assembler generates a symbolic address. A register name specifies the contents of the named register. This are examples of instructions that use operands without prefixes:

```
Label: ADD 0FFFFh,R5    ; add contents of absolute address to register
      ADD Label,R5      ; add contents of symbolic address to register
```

- **& prefix — the operand is an absolute address.** If you use the & sign as a prefix, the assembler treats the operand as an absolute address, similar to using no prefix. The operand has to specify a constant value:

```
      MOV &200h,R5
      MOV 200h,R5
```

Both instructions generate the same object code, moving the contents of absolute address 200h to register R5.

- **# prefix — the operand is an immediate value.** If you use the # sign as a prefix, the assembler treats the operand as an immediate value. This is true even when the operand is an address; the assembler treats the address as a value instead of using the contents of the address. This is an example of an instruction that uses an operand with the # prefix:

```
Label: ADD #123,R5
```

The operand #123 is an immediate value. The assembler adds 123 (decimal) to the contents of register R5.

- **@ prefix — the operand is an indirect address.** If you use the @ sign as a prefix, the assembler treats the operand as an indirect address; that is, it uses the contents of the operand as an address. This is an example of an instruction that uses an operand with the @ prefix:

```
Label: MOV @R4,R4
```

The operand @R4 specifies an indirect address. The assembler goes to the address specified by the contents of register R4 and then moves the contents of that location to register R4.

- **Immediate Addressing for Directives**

The immediate addressing mode is used mostly with instructions; in some cases, it can also be used with the operands of directives.

Usually, it is not necessary to use the immediate addressing mode for directives. Compare the following statements:

```
ADD #10, R4
.byte 10
```

In the first statement, the immediate addressing mode is necessary to tell the assembler to add the value 10 to register R4. In the second statement, however, immediate addressing is not used; the assembler expects the operand to be a value and initializes a byte with the value 10.

Comment Field

A comment can begin in any column and extends to the end of the source line. A comment can contain any ASCII character, including blanks. Comments are printed in the assembly source listing, but they do not affect the assembly.

A source statement that contains only a comment is valid. If it begins in column 1, it can start with a semicolon (;) or an asterisk (*). Comments that begin anywhere else on the line **must** begin with a semicolon. The asterisk identifies a comment only if it appears in column 1.

3.5 Constants

The assembler supports six types of constants:

- Binary integer constants
- Octal integer constants
- Decimal integer constants
- Hexadecimal integer constants
- Character constants
- Assembly-time constants

The assembler maintains each constant internally as a 32-bit quantity. Note that constants **are not sign extended**. For example, the constant 0FFH is equal to 00FF (base 16) or 255 (base 10); it **does not** equal -1.

Binary Integers

A binary integer constant is a string of up to 16 binary digits (0s and 1s) followed by the suffix **B** (or **b**). If fewer than 16 digits are specified, the assembler right-justifies the value and zero-fills the unspecified bits. These are examples of valid binary constants:

00000000B **Constant equal to 0₁₀ or 0₁₆**
0100000b **Constant equal to 32₁₀ or 20₁₆**
01b **Constant equal to 1₁₀ or 1₁₆**
11111000B **Constant equal to 248₁₀ or 0F8₁₆**

Octal Integers

An octal integer constant is a string of up to 6 octal digits (0 through 7) followed by the suffix **Q** (or **q**). These are examples of valid octal constants:

10Q **Constant equal to 8₁₀ or 8₁₆**
100000Q **Constant equal to 32,768₁₀ or 8000₁₆**
226Q **Constant equal to 150₁₀ or 96₁₆**

Decimal Integers

A decimal integer constant is a string of decimal digits, ranging from? -32,768 to 65,535. These are examples of valid decimal constants:

1000 **Constant equal to 1000₁₀ or 3E8₁₆**
-32768 **Constant equal to -32,768₁₀ or 8000₁₆**
25 **Constant equal to 25₁₀ or 19₁₆**

Hexadecimal Integers

A hexadecimal integer constant is a string of up to 4 hexadecimal digits followed by the suffix **H** (or **h**). Hexadecimal digits include the decimal values 0-9 and the letters A-F or a-f. A

hexadecimal constant must begin with a decimal value (0-9). If fewer than 4 hexadecimal digits are specified, the assembler right-justifies the bits. These are examples of valid hexadecimal constants:

78h **Constant equal to 120₁₀ or 0078₁₆**
0Fh **Constant equal to 15₁₀ or 000F₁₆**
37ACh **Constant equal to 14,252₁₀ or 37AC₁₆**

Character Constants

A character constant is a single character enclosed in *single* quotes. The characters are represented internally as 8-bit ASCII characters. Two consecutive single quotes are required to represent each single quote that is part of a character constant. A character constant consisting only of two single quotes is valid and is assigned the value 0. These are examples of valid character constants:

'a' **Defines the character constant *a* and is represented internally as 61₁₆**
'C' **Defines the character constant *C* and is represented internally as 43₁₆**
"" **Defines the character constant "" and is represented internally as 27₁₆**
" **Defines a null character and is represented internally as 00₁₆**

Note the difference between character *constants* and character *strings*. A character constant represents a single integer value; a string is a list of characters.

Assembly-Time Constants

If you use the `.equ` directive to assign a value to a symbol, the symbol becomes a constant. In order to use this constant in expressions, the value that is assigned to it must be absolute. For example:

```
sym        .equ     3  
          MOV #sym,R10
```

You can also use the `.equ` directive to assign symbolic constants for register names. In this case, the symbol becomes a synonym for the register:

```
sym        .equ     R14  
          MOV #10,sym
```

3.6 Character Strings

A character string is a string of characters enclosed in *double* quotes. Double quotes that are part of character strings are represented by two consecutive double quotes. The maximum length of a string varies and is defined for each directive that requires a character string. Characters are represented internally as 8-bit ASCII characters. Appendix lists valid characters.

These are examples of valid character strings:

"sample program" **defines a 14-character string, sample program**

"PLAN ""C"" **defines an 8-character string, PLAN "C"**

Character strings are used for the following:

- Filenames, as in .copy "filename"
- Section names, as in .sect "section name"
- Data initialization directives, as in .byte "charstring"
- Operand of .string or .byte directive

3.7 Symbols

Symbols are used as labels, constants, and substitution symbols. A symbol name is a string of up to 32 alphanumeric characters (A-Z, a-z, 0-9, \$, _and?). The first character in a symbol cannot be a number; symbols cannot contain embedded blanks. The symbols you define are case sensitive; for example, the assembler recognizes *ABC*, *Abc*, and *abc* as three unique symbols. You can override case sensitivity with the -c assembler option. This type of symbol is valid only during the assembly in which it is defined, unless you use the .global directive to declare it as an external symbol.

Labels

Symbols that are used as labels become symbolic addresses that are associated with locations in the program. A label used locally within a file must be unique. Mnemonic op-codes and assembler directive names (without the '_' prefix) are valid label names.

Labels can also be used as the operand of a .global, .ref, .def, or .bss directive; for example:

```
.global    label1

label2  nop
        mov     label1, R4
        br      label2
```

Local Labels

Local labels are a special type of label whose scope and effect are only temporary. A local label has the form \$*n*, where *n* is a decimal digit in the range 0-9. For example, \$4 and \$1 are valid local labels.

Normal labels must be unique (they can be declared only once), and they can be used as constants in the operand field. Local labels, however, can be undefined and defined again. If a local label is used as an operand, it can be used only as an operand for a 10-bit jump instruction.

A local label can be undefined, or reset, in one of four ways:

- By the .newblock directive
- By changing sections (using a .sect, .text, or .data directive)
- By entering an include file (specified by the .include or .copy directive)
- By leaving an include file (specified by the .include or .copy directive)

This is an example of code that declares and uses a local label legally:

```
Label1:      mov R12,R13
             jnz $1
             mov #-1,R13
$1           cmp R13,R4
             .newblock; Undefine $1 so it can be used again
             jne $1
             inc R13
$1           add R13,R14
```

The following code uses a local label illegally:

```
Label1:      mov R12,R13
             jnz $1
             mov #-1,R13
$1           cmp R3,R4
             jne $1
             inc R13
$1           add R13,R14 ; WRONG - $1 is multiply defined
```

Local labels are especially useful in macros. If a macro contains a normal label and is called more than once, the assembler issues a multiple-definition error. However, if you use a local label within a macro and then use `.newblock` within the macro, the local label is used and reset each time the macro is expanded.

Up to ten local labels can be in effect at one time. After you undefine a local label, you can define it and use it again. Local labels do not appear in the object code symbol table.

Constants

Symbols can be set to constant values. By using constants, you can equate meaningful names with constant values. The `.equ`, `.set`, and `.struct/.tag/.endstruct` directives enable you to set constants to symbolic names. Symbolic constants **cannot** be redefined. The following example shows how these directives can be used:

```
K           .set      1024                ; constant definitions
maxbuf      .set      2*K

item        .struct                    ; item structure definition
             .byte    value              ; constant offsets value = 0
             .byte    delta              ; constant offsets delta = 1
i_len       .endstruct

array       .tag      item                ; array declaration
             .bss     array, i_len*K

             MOV      array.delta,R4      ; array+1
```

The assembler also has several predefined symbolic constants; these are discussed in the next subsection.

Symbolic Constants

The assembler has several predefined symbols, including the following:

- **\$**, the dollar sign character, represents the current value of the section program counter (SPC).
- **Register symbols**, which are of the form Rn or rn , where n is an expression that evaluates in the range 0-15. (If the number is greater than 15, the symbol is not considered a register symbol.) The number may be decimal. Note that **PC**, **SP** and **SR** are valid register symbols; they represent registers with special functions (R0 - R3).

Substitutions Symbols

Symbols can be assigned a string value (variable). This enables you to alias character strings by equating them to symbolic names. Symbols that represent character strings are called substitution symbols. When the assembler encounters a substitution symbol, its string value is substituted for the symbol name. Unlike symbolic constants, substitution symbols **can** be redefined.

A string can be assigned to a substitution symbol anywhere within a program; for example:

```
.asg    "R13",    SP1
.asg    "+",      pls
.asg    "-5",     min5

ADD     # min5,SP1
```

When you are using macros, substitution symbols are important because macro parameters are actually substitutions symbols that are assigned a macro argument. The following code shows how substitution symbols are used in macros:

```
add2    .macro     src,dest    ;  add2 macro definition

        mov       src,R4
        mov       R4,R5
        mov       dest,R4
        add       R5,R4
        mov       R4,dest

        .endm

*add2 invocation
        add2      loc1, loc2
```

3.8 Expressions

An expression is a constant, a symbol, or a series of constants and symbols separated by arithmetic operators. The range of valid expression values is -32,768 to 65,535. These are the three main factors that influence the order of expression evaluation:

Parentheses	Expressions that are enclosed in parentheses are always evaluated first. $8/(4/2) = 4$, but $8/4/2 = 1$ Note that you cannot substitute braces ({ }) or brackets ([]) for parentheses.
Precedence groups	Operators, listed in the next table, are divided into nine precedence groups. When the order of expression evaluation is not determined by parentheses, the highest precedence operation is evaluated first. $8 + 4/2 = 10$ ($4/2$ is evaluated first)
Left-to-right evaluation	When parentheses and precedence groups do not determine the order of expression evaluation, the expressions are evaluated from left to right. Note that the highest precedence group is evaluated from right to left. $8/4*2 = 4$, but $8/(4*2) = 1$

3.8.1 Operators

Group	Operator	Description
1	+ - ~ !	Unary plus Unary minus 1s complement Logical NOT
2	* / %	Multiplication Division Modulo
3	+ -	Addition Subtraction
4	<< >>	Shift left Shift right
5	< <= > >=	Less than Less than or equal to Greater than Greater than or equal to
6	= (==) != (<>)	Equal to Not equal to
7	&	Bitwise AND
8	^	Bitwise XOR
9		Bitwise OR

- Notes:**
- 1) Operators in parentheses () indicate an alternate form.
 - 2) Group 1 operators are evaluated right to left. All other operators are evaluated left to right.

Table 3.1: Operators Used in Expressions (Precedence)

3.8.2 Expression Overflow and Underflow

The assembler checks for overflow and underflow conditions when arithmetic operations are performed at assembly time. The assembler issues a Value Truncated warning whenever an overflow or underflow occurs. The assembler **does not** check for overflow or underflow in multiplication.

3.8.3 Well-Defined Expressions

Some assembler directives require well-defined expressions as operands. Well-defined expressions contain only symbols or assembly-time constants that are defined before they are encountered in the expression. The evaluation of a well-defined expression must be absolute.

This is an example of a well-defined expression:

1000h+X

where X was previously defined as an absolute symbol.

3.8.4 Conditional Expressions

The assembler supports relational operators that can be used in any expression; they are especially useful for conditional assembly. Relational operators include the following:

=	Equal to	==	Equal to
!=	Not equal to	<>	Not Equal to
<	Less than	<=	Less than or equal to
>	Greater than	>=	Greater than or equal to

Conditional expressions evaluate to 1 if true and 0 if false, and may be used only on operands of equivalent types, e.g., absolute value compared to absolute value, but not absolute value compared to relocatable value.

3.8.5 Relocatable Symbols and Legal Expressions

The following table summarizes valid operations on absolute, relocatable, and external symbols. An expression cannot multiply or divide by a relocatable or external symbol. An expression cannot contain unresolved symbols that are relocatable with respect to different sections.

Symbols that have been defined as global with the .global directive can also be used in expressions; in the table, these symbols are referred to as *external*.

If A is...	If B is...	A + B is...	A - B is...
absolute	absolute	absolute	absolute
absolute	relocatable	relocatable	illegal
absolute	external	external	illegal
relocatable	absolute	relocatable	relocatable
relocatable	relocatable	illegal	absolute *
relocatable	external	illegal	illegal
external	absolute	external	external
external	relocatable	illegal	illegal
external	external	illegal	illegal

* A and B must be in the same section; otherwise, this is illegal.

Table 3.2: Expressions With Absolute and Relocatable Symbols

Here are some examples of expressions that use relocatable and absolute symbols. These examples use four symbols that are defined in the same section:

```

                .global extern_1    ;Defined in an external module
intern_1:      .word 'D'            ;Relocatable, defined in current module
LAB1:         .equ 2                ;LAB1 = 2
intern_2:      ;Relocatable, defined in current module

```

- Example 1:**

The first statement in this example puts the value 51 into register R4. The second statement puts the value 27 into register R4.

```

MOV      #(LAB1 + (4+3) * 7), R4      ; R4 = 51
MOV      #(LAB1 + 4 + 3 * 7), R4      ; R4 = 27

```

- Example 2**

All legal expressions can be reduced to one of two forms:

relocatable symbol ± absolute symbol

or

absolute value

Unary operators can be applied only to absolute values; they cannot be applied to relocatable symbols. Expressions that cannot be reduced to contain only one relocatable symbol are illegal. The first statement in the following example is legal; the others are illegal.

```

MOV  extern_1 - 10, R4      ; Legal
MOV  10-extern_1, R4        ; Can't negate relocatable symbol
MOV  -(intern_1), R4        ; Can't negate relocatable symbol
MOV  extern_1/10, R4        ; / is not an additive operator
MOV  intern_1 + extern_1,R4 ; Multiple relocatables

```

- **Example 3**

The first statement below is legal; although `intern_1` and `intern_2` are relocatable, their difference is absolute because they're in the same section. Subtracting one relocatable symbol from another reduces the expression to *relocatable symbol + absolute value*. The second statement is illegal because the sum of two relocatable symbols is not an absolute value.

```
MOV      intern_1 - intern_2 + extern_1, R4    ; (legal)
MOV      intern_1 + intern_2 + extern_1, R4    ; (illegal)
```

- **Example 4**

An external symbol's placement in an expression is important to expression evaluation. Although the statement below is similar to the first statement in the previous example, it is illegal. This is because of left-to-right operator precedence; the assembler attempts to add `intern_1` to `extern_1`.

```
MOV      intern_1 + extern_1 - intern_2, R4    ; (illegal)
```

3.9 Source Listings

A source listing shows source statements and the object code they produce. To obtain a listing file, invoke the assembler with the -l (lowercase "l") option.

At the top of each source listing page are two banner lines, a blank line, and a title line. Any title supplied by a .title directive is printed on this line; a page number is printed to the right of the title. If you don't use the .title directive, the title area is left blank. The assembler inserts a blank line below the title line.

Each line in the source file may produce a line in the listing file that shows a source statement number, an SPC value, the object code assembled, and the source statement. A source statement may produce more than one byte of object code and may be listed on more than one line. If so, each additional line is listed immediately following the source statement line.

Field 1 Source Statement Number

Line Number

The source statement number is a decimal number. The assembler numbers source lines as it encounters them in the source file; some statements increment the line counter but are not listed. (For example, .title statements and statements following a .nolist are not listed.) The difference between two consecutive source line numbers indicates the number of statements in the source file that are not listed.

Include File Letter

The assembler may precede a line with a letter; the letter indicates that the line is assembled from an include file.

Nesting Level Number

The assembler may precede a line with a number; the number indicates the nesting level of macro expansions and loop blocks.

Field 2 Section Program Counter

This field contains the section program counter (SPC) value (hexadecimal). All sections (.text, .data, .bss, and named sections) maintain separate SPCs. Some directives do not affect the SPC; they leave this field blank.

Field 3 Object Code

This field contains the hexadecimal representation of the object code. All machine instructions and directives use this field to list object code. This field also contains two columns immediately preceding the object code, which indicate additional information about the line of object code.

The first column either will be blank or will contain an asterisk (*). The asterisk indicates that the object code is not a direct mapping from the assembly source.

The second column indicates a relocation type that is associated with one of the operands for this line of source code. If more than one operand is relocatable, this column indicates the relocation type for the first one. The characters that may appear in this column and their associated relocation types are illustrated in the following table:


!	external reference (global)	"	.data relocatable
'	text relocatable	-	.bss, .usect relocatable
+	.sect relocatable		

Field 4 Source Statement Field


This field contains the characters of the source statement as they were scanned by the assembler. The assembler accepts a maximum line length of 200 characters. Spacing in this field is determined by the spacing in the source statement.

Example of an assembler listing with each of the four fields identified:

1			.global func3
2	0000		.bss data1,1
3	0001		.bss data2,1
4			
5			.copy "mac1.inc"
1		add2	.macro src, dest
2			mov src, R4
3			mov R4, R5
4			mov dest, R4
5			add R5,R4
6			mov R4, dest
7			.endm
8			
9			*****
10			** interrupt vectors **
11			*****
12	0000		.sect "int_vecs"
13	0000	'0000	.word func1
14	0002	+0000	.word func2
15	0004	!0000	.word func3
16			
17			*****
18			** .text section **
19			*****
20	0000		.text
21	0000		
22	0000	func1:	add2 data1, data2
1	0000	-40140000	mov data1, R4
1	0004	4405	mov R4, R5
1	0006	-4014fff9	mov data2, R4
1	000a	5504	add R5,R4
1	000c	-4480fff3	mov R4, data2
33	0020	-40140000	mov data1, R4
34	0024	940a	cmp R4,R10
35	0026	'3401	jge lab
36	0028	4304	clr R4
37	002a	1300	lab: reti
38			
39	0000		.sect "other_code"
40	0000	9405	func2 cmp R4,R5
41	0002	1300	reti



Field 1 Field 2 Field 3



Field 4

Example 3.1: An Assembler Listing

3.10 Cross–Reference Listings

A cross–reference listing shows symbols and their definitions. To obtain a cross–reference listing, invoke the assembler with the -x option or use the .option directive. The assembler will append the cross–reference to the end of the source listing.

LABEL	VALUE	DEFN	REF
data1	0000 -	2	27 28
data2	0001 -	3	27 27*
func1	0000 `	26	18
func2	0000 +	40	19
func3	REF		1 20
lab	002a `	37	35

Example 3.2: An Assembler Cross–Reference Listing

LABEL	column contains each symbol that was defined or referenced during the assembly.
VALUE	column contains a 4–digit hexadecimal number, which is the value assigned to the symbol <i>or</i> a name that describes the symbol's attributes. A value may also be preceded by a character that describes the symbol's attributes. The next table lists these characters and names.
DEFINITION	(DEFN) column contains the statement number that defines the symbol. This column is blank for undefined symbols.
REFERENCE	(REF) column lists the line numbers of statements that reference the symbol. If the line number is followed by an asterisk (*), that reference may modify the contents of the object. A blank in this column indicates that the symbol was never used.

Character or Name	Meaning
REF	External reference (global symbol)
UNDF	Undefined
`	Symbol defined in a .text section
"	Symbol defined in a .data section
+	Symbol defined in a .sect section
-	Symbol defined in a .bss or .usect section

Table 3.3: Symbol Attributes