# CodeWarrior™ Development Studio for Freescale™ DSP56800x EmbeddedSystems Assembler Manual

Revised: May 7, 2006

freescale™ semiconductor

## How to Contact Freescale

| | |
|---|---|
| Corporate Headquarters | Freescale Corporation<br>7700 West Parmer Lane<br>Austin, TX 78729<br>U.S.A. |
| World Wide Web | http://www.freescale.com/codewarrior |
| Technical Support | http://www.freescale.com/support |

# Table of Contents

# 1

# Introduction

The Freescale DSP Assembler not only processes assembly code for the Freescale DSP instruction set, but offers a set of useful directives and macros which makes your assembly coding much easier. This manual discusses the directives and macros.

This introductory chapter provides this information:

- Assembler Preferences Panel on page 9
- Differences from ASM56800 on page 9
- Where to Read More on page 10

## Assembler Preferences Panel

A DSP-specific preferences panel in the CodeWarrior IDE overlaps with the functions of some of the `OPT` assembler directives. CodeWarrior targeting documentation for Freescale DSP fully describes this preference panel. In the case of a panel preference and `OPT` directive conflict, the directive takes precedence. For more information, see the targeting documentation.

## Differences from ASM56800

The CodeWarrior Freescale DSP Assembler provides very similar function as the `asm56800` assembler from Freescale Semiconductor, Inc. However, there are some differences which are outlined below.

Unimplemented or changed directives are:

- `ORG` (partially supported)
- `COBJ`
- `IDENT`
- `SYMOBJ`
- `MACLIB`

## Where to Read More

This manual does not discuss how to construct programs for the Freescale DSP digital signal processor.

For a complete description of the Freescale DSP instruction set, consult the following:

- *DSP56800E 16-bit Digital Signal Processor Core Manual*
  Freescale Semiconductor, Inc. 2001. Part DSP56800ERM/D.
- *DSP56800 16-bit Digital Signal Processor Family Manual*
  Freescale Semiconductor, Inc. 1996. Part DSP56800FM/AD.

These references are available at the following web address:

`http://www.Freescale.com`

PRODUCT NAME specifies ELF (Executable and Linker Format) as the output file format, and DWARF as the symbol file format. For more information about those file formats, you should read the following documents:

- *Executable and Linker Format, Version 1.1*, published by UNIX System Laboratories.
- *DWARF Debugging Information Format, Revision: Version 1.1.0*, published by UNIX International, Programming Languages SIG, October 6, 1992.
- *DWARF Debugging Information Format, Revision: Version 2.0.0*, Industry Review Draft, published by UNIX International, Programming Languages SIG, July 27, 1993.

2

# Assembler Statement Syntax

Assembler statements are used to control the operation of the assembler itself. The syntax of assembler statements mirrors the syntax of assembly language. This chapter is organized into the following topics:

## Statement Format

Motorola DSP Assembler statements are formatted as follows:

*labeloperationoperandX fieldY field*          *Comment*

The following syntax applies:

- Spaces or tabs delimit the fields.
- Comments normally follow the operand or Y field and start with a semi-colon (;). Lines which consist only of space or tab characters are treated as comments. Any line that starts with a semi-colon is a comment.

### Label field

The first field on the source line is the label field. The label field follows these rules:

- A space or tab as the first character indicates that there is no label for this line.
- An alphabetic character as the first character indicates that the line contains a symbol called label.
- An underscore character as the first character indicates that the label is a local label.

A label may be indented (not starting with the first character of the line) if the last character in the label is a colon (:). Local labels have a scope limited by any two bounding non-local labels. Local labels are useful in the instance that a label is needed, but you don't need to refer to it outside of a limited scope, such as in the case of a `DO` loop.

You may not define a label twice, unless you do so in conjunction with a `SECTION`, a `SET` directive, or as a local label.

### Operation field

The entries in the operation field may be one the following types:

- Opcode - instructions for the Motorola DSP processor.
- Directive - a special code that controls the assembly process.
- Macro call - invocation of a previously defined macro.

The assembler first searches for a macro with the same name as the operation specified in this field. Then the assembler searches the instruction set and lastly the assembler directives. Therefore, macro names can replace machine instructions. Be careful not to name macros after instruction names unknowingly. You can use the `REDIRECT` directive to change this default behavior.

### Operand field

The contents of the operand field are dependent on the contents of the operation field. The operand may contain a symbol, an expression, or a combination of symbols and expressions separated by commas.

### Data Transfer fields (X and Y Fields)

Most opcodes can specify one or more data transfers to occur during the execution of the instruction. These data transfers are indicated by two addressing mode operands separated by a comma, with no embedded blanks. See “Expression Memory Space Attribute” on page 13 on page 13 for further discussion of Motorola DSP data transfers.

### Comment field

Comments begin with a semicolon (;). All characters after the semicolon are ignored.

## Name and Label Format

Identifiers or symbol names may be as long as 512 characters. The first character of the name must be alphabetic; remaining characters may be alphanumeric or an underscore character (_). By default, upper- and lowercase characters are distinct unless case-sensitivity is turned off in the Motorola DSP Assembler settings panel.

The identifiers listed in Table Table 2.1 on page 13 are reserved for the assembler. These names identify the Motorola DSP registers.

**Table 2.1 Reserved Identifiers**

| | | |
|---|---|---|
| X | A | PC |
| X0 | A0 | M01 |
| X1 | A1 | N |
| Y | A2 | SP |
| Y0 | B | MR |
| Y1 | B0 | CCR |
| R0 | B1 | SR |
| R1 | B2 | OMR |
| R2 | LC | SSH |
| R3 | LA | |

## Expressions

The Motorola DSP Assembler supports the following constant types and other expressions.

### Absolute and Relative Expressions

An expression may be either relative or absolute. An absolute expression is one which consists only of absolute terms.

A relative expression consists of a relative mode. All address expressions must adhere to these definitions for absolute or relative expressions. This is because only these types of expressions retain a meaningful value after program relocation.

### Expression Memory Space Attribute

Memory space attributes become important when an expression is used as an address. Errors occur when the memory space attribute of the expression result does not match the explicit or implicit memory space specified in the source code. Memory spaces are explicit when the address has any of the following forms:

`X:`*address expression*

`P:`*address expression*

The memory space is implicitly P when an address is used as the operand of a `DO`, branch, or jump-type instruction.

Expressions used for immediate addressing can have any memory space attribute.

## Constants

Constants represent data that does not change during the course of the program.

### Numeric Constants

Binary and hexadecimal constants require the use of the leading radix indicator character:

- Binary—Percent sign (%) followed by a string of binary digits (0,1). Example: `%0101`
- Hexadecimal—Dollar sign ($) followed by a string of hexadecimal digits (0-9, A-F, a-f). Example: `$7FFF`
- Decimal—A string of digits (0-9), optionally preceded by a grave accent (`` ` ``). Example: `` `12345 ``

A constant may be expressed without the leading radix indicator if a `RADIX` directive is used before the constant evaluation.

### String Constants

String constants used in expressions are converted to a concatenated sequence of right-aligned ASCII bytes. String constants for the Motorola DSP processor are limited to four characters; subsequent characters are ignored by the assembler. You may use the `DC` and `DCB` directives to define strings longer than 4 bytes. Table 2.2 on page 14 shows some examples.

**Table 2.2 Use of String Constants in Expressions**

| Expression | String | Bytes |
|---|---|---|
| `'ABCD'` | ABCD | `$41424344` |
| `'"79'` | "79 | `$00273739` |

**Table 2.2  Use of String Constants in Expressions (*continued*)**

| Expression | String | Bytes |
|---|---|---|
| `''` | null string | `$00000000` |
| `'abcdef'` | abcd | `$61626364` |

## Operators

This section defines valid operators and operator precedence.

Valid operators:

- Unary Operators on page 15
- Shift Operators on page 15
- Bitwise Operators on page 15
- Arithmetic Operators on page 16
- Relational Operators on page 16
- Logical Operators on page 16

**Table 2.3  Unary Operators**

| | |
|---|---|
| positive | + |
| negative | - |
| one’s complement | ~ |
| logical negate | ! |

**Table 2.4  Shift Operators**

| | |
|---|---|
| shift left | << |
| shift right | >> |

**Table 2.5  Bitwise Operators**

| | |
|---|---|
| AND | & |

**Table 2.5 Bitwise Operators (*continued*)**

| | |
|---|---|
| OR | \| |
| XOR | ^ |

**Table 2.6 Arithmetic Operators**

| | |
|---|---|
| multiplication | * |
| division | / |
| modulus | % |
| addition | + |
| subtraction | - |

**Table 2.7 Relational Operators**

| | |
|---|---|
| less than | < |
| less than or equal | <= |
| greater than | > |
| greater than or equal | >= |
| equal | == |
| not equal | != |

**Table 2.8 Logical Operators**

| | |
|---|---|
| logical AND | && |
| logical OR | \|\| |

## Operator Precedence

Operators are evaluated in the following order:

1. parenthetical expression (innermost first)
2. unary positive, unary negative, ~, !

3. <<, >>
4. &, |, ^
5. multiplication, division, modulus
6. addition, subtraction
7. <, <=, >, >=, ==, !=
8. &&, ||

Operators with the same precedence are evaluated left to right.

# Functions

The Motorola DSP Assembler provides a number of convenient functions for the programmer. There are five different types of functions:

- Mathematical functions on page 17
- Conversion functions on page 25
- String functions on page 29
- Macro functions on page 30
- Assembler Mode functions on page 32

## Mathematical functions

This section describes the Motorola DSP Assembler mathematical functions:

- `ABS` on page 18
- `ACS` on page 18
- `ASN` on page 19
- `AT2` on page 19
- `ATN` on page 19
- `CEL` on page 20
- `COH` on page 20
- `COS` on page 20
- `FLR` on page 21
- `L10` on page 21
- `LOG` on page 21
- `MAX` on page 22
- `MIN` on page 22

## ABS

Absolute value

`@ABS(`*expression*`)`

### Description

Returns the absolute value of *expression* as a floating-point value.

### Example

```
MOVE      #@ABS(0.5),b0
```

## ACS

Arc cosine

`@ACS(`*expression*`)`

### Description

Returns the arc cosine of *expression* as a floating-point value in the range zero to pi. The result of *expression* must be between -1 and 1.

### Example

```
ACOS      =      @ACS(-1.0)      ; ACOS = 3.141593
```

## ASN

Arc sine

`@ASN(`*expression*`)`

### Description

Returns the arc sine of *expression* as a floating-point value in the range -pi/2 to pi/2. The result of *expression* must be between -1 and 1.

### Example

```
ARCSINE     SET @ASN(-1.0)      ; ARCSINE = -1.570796
```

## AT2

Arc tangent

`@AT2(`*expression1*, *expression2*`)`

### Description

Returns the arc tangent of *expression1*/*expression2* as a floating-point value in the range -pi to pi. *expression1* and *expression2* must be separated by a comma.

### Example

```
ATAN EQU @AT2(-1.0,1.0)      ; ATAN = -0.7853982
```

## ATN

Arc tangent

`@ATN(`*expression*`)`

### Description

Returns the arc tangent of *expression* as a floating-point value in the range -pi/2 to pi/2.

### Example

```
MOVE #@ATN(1.0),b0
```

## CEL

Ceiling

`@CEL(`*expression*`)`

### Description

Returns a floating-point value which represents the smallest integer greater than or equal to *expression.*

### Example

```
CEIL     SET @CEL(-1.05)     ; CEIL = -1.0
```

## COH

Hyperbolic cosine

`@COH(`*expression*`)`

### Description

Returns the hyperbolic cosine of *expression* as a floating-point value.

### Example

```
HYCOS EQU @COH(0.5)
```

## COS

Cosine

`@COS(`*expression*`)`

### Description

Returns the cosine of *expression* as a floating-point value.

### Example

```
DC      -@COS(@CVF(0.5)*0.5)
```

## FLR

Floor

`@FLR(`*expression*`)`

### Description

Returns a floating-point value which represents the largest integer less than or equal to *expression*.

### Example

```
FLOOR SET @FLR(2.5)      ; FLOOR = 2.0
```

## L10

Log base 10

`@L10(`*expression*`)`

### Description

Returns the base 10 logarithm of *expression* as a floating-point value. The *expression* must be greater than zero.

### Example

```
LOG EQU @L10(100.0)      ; LOG = 2
```

## LOG

Natural logarithm

`@LOG(`*expression*`)`

### Description

Returns the natural logarithm of *expression* as a floating-point value. The *expression* must be greater than zero.

### Example

```
LOG EQU @LOG(100.0)      ; LOG = 4.605170
```

## MAX

Maximum value

`@MAX(`*expression1[,...,expressionN]*`)`

### Description

Returns the greatest of *expression1,...,expressionN* as a floating-point value.

### Example

```
MAX DC @MAX(1.0,5.5,-3.25)     ; MAX = 5.5
```

## MIN

Minimum value

`@MIN(`*expression1[,...,expressionN]*`)`

### Description

Returns the least of *expression1,...,expressionN* as a floating-point value.

### Example

```
MIN DC @MIN(1.0,5.5,-3.25)     ; MIN = -3.25
```

## POW

Raise to a power

`@POW(`*expression1,expression2*`)`

### Description

Returns *expression1* raised to the power *expression2* as a floating-point value. The two expressions must be separated by a comma.

### Example

```
BUF EQU @CVI(@POW(2.0,3.0))     ; BUF = 8
```

## `RND`

Random value

`@RND()`

### Description

Returns a random value in the range 0.0 to 1.0.

### Example

```
SEED DC @RND()
```

## `SGN`

Return sign

`@SGN(`*expression*`)`

### Description

Returns the sign of *expression* as an integer: -1 if the argument is negative, 0 if zero, 1 if positive.

### Example

```
IF @SGN(0.5)      ; is input positive?
```

## `SIN`

Sine

`@SIN(`*expression*`)`

### Description

Returns the sine of *expression* as a floating-point value.

### Example

```
DC @SIN(@CVF(0.5)*0.5)
```

## SNH

Hyperbolic sine

`@SNH(`*expression*`)`

### Description

Returns the hyperbolic sine of *expression* as a floating-point value.

### Example

```
HSINE EQU @SNH(0.5)
```

## SQT

Square root

`@SQT(`*expression*`)`

### Description

Returns the square root of *expression* as a floating-point value. The *expression* must be positive.

### Example

```
SQRT EQU @SQT(3.5)     ; SQRT = 1.870829
```

## TAN

Tangent

`@TAN(`*expression*`)`

### Description

Returns the tangent of *expression* as a floating-point value.

### Example

```
MOVE #@TAN(1.0),b0
```

### TNH

Hyperbolic tangent

`@TNH(`*expression*`)`

#### Description

Returns the hyperbolic tangent of *expression* as a floating-point value.

#### Example

```
HTAN = @TNH(VAL)
```

### XPN

Exponential

`@XPN(`*expression*`)`

#### Description

Returns the exponential function (base e raised to the power of *expression*) as a floating point value.

#### Example

```
EXP EQU @XPN(1.0)      ; EXP = 2.718282
```

## Conversion functions

This section describes the DSP56800 Assembler conversion functions:

- CVF on page 26
- CVI on page 26
- CVS on page 26
- FLD on page 27
- FRC on page 27
- LFR on page 27
- LNG on page 28
- RVB on page 28

- UNF on page 28

## CVF

Convert integer to floating point

`@CVF(`*expression*`)`

### Description

Converts the result of *expression* to a floating-point value.

### Example

```
FLOAT SET @CVF(5)      ; FLOAT = 5.0
```

## CVI

Convert floating point to integer

`@CVI(`*expression*`)`

### Description

Converts the result of *expression* to an integer value. This function should be used with caution, since the conversions can be inexact (e.g., floating-point values are truncated).

### Example

```
INT SET @CVI(-1.05)      ; INT = -1
```

## CVS

Convert memory space

`@CVS({X|P},`*expression*`)`

### Description

Converts the memory space attribute of *expression* to that specified by the first argument; returns *expression*.

### Example

```
LOADDR EQU @CVS(X,TARGET)
```

## FLD

Shift and mask operation

`@FLD(`*base,value,width[,start]*`)`

### Description

Shift and mask *value* into *base* for *width* bits beginning at bit *start*. If *start* is omitted, zero (least significant bit) is assumed. All arguments must be positive integers and none may be greater than the target word size.

### Example

```
SWITCH EQU @FLD(TOG,1,1,7) ; turn eighth bit on
```

## FRC

Convert floating-point to fractional

`@FRC(`*expression*`)`

### Description

Performs scaling and convergent rounding to obtain the fractional representation of the floating-point *expression* as an integer.

### Example

```
FRAC EQU @FRC(1.0)+1
```

## LFR

Convert floating-point to long fractional

`@LFR(`*expression*`)`

### Description

Performs scaling and convergent rounding to obtain the fractional representation of the floating-point *expression* as a long integer.

### Example

```
LFRAC EQU @LFR(1.0)
```

## LNG

Concatenate to double-word

`@LNG(`*expression1,expression2*`)`

### Description

Concatenates the single-word *expression1* and *expression2* into a double-word value such that *expression1* is the high word and *expression2* is the low word.

### Example

```
LWORD DC @LNG(5.0,2.0)
```

## RVB

Reverse bits in field

`@RVB(`*expression1[,expression2]*`)`

### Description

Reverse the bits in *expression1* delimited by the number of bits in *expression2*. If *expression2* is omitted, the field is bounded by the target word size. Both expressions must be single-word integer values.

### Example

```
REV EQU @RVB(0.5) ; reverse all bits in value
```

## UNF

Convert fractional to floating-point

`@UNF(`*expression*`)`

### Description

Converts *expression* to a floating-point value. The *expression* should represent a binary fraction.

### Example

```
FRC EQU @UNF($400000)      ; FRC = 0.5
```

### String functions

This section describes the Motorola DSP Assembler string functions:

- LEN on page 29
- POS on page 29
- SCP on page 30

## LEN

Length of string

`@LEN(`*string*`)`

### Description

Returns the length of *string* as an integer.

### Example

```
SLEN SET @LEN('string') ; SLEN = 6
```

## POS

Position of substring in string

`@POS(`*string1,string2[,start]*`)`

### Description

Returns the position of *string2* in *string1* as an integer, starting at position *start*. If *start* is not given, the search begins at the beginning of *string1*. If the *start* argument is specified, it must be a positive integer and cannot exceed the length of the source string.

### Example

```
ID EQU @POS('DSP96000','96') ; ID = 3
```

## SCP

String compare

`@SCP(`*string1,string2*`)`

### Description

Returns an integer 1 if the two strings compare, 0 otherwise. The two strings must be separated by a comma.

### Example

```
IF @SCP('main','MAIN')      ; does main equal MAIN?
```

### Macro functions

This section describes the Motorola DSP Assembler macro functions:

- ARG on page 30
- CNT on page 31
- MAC on page 31
- MXP on page 31

## ARG

Macro argument function

`@ARG(`*symbol* | *expression*`)`

### Description

Returns integer 1 if the macro argument represented by *symbol* or *expression* is present, 0 otherwise. If the argument is a symbol, it must be single-quoted and refer to a dummy argument name. If the argument is an expression, it refers to the ordinal position of the argument in the macro dummy argument list. A warning is issued if this function is used when no macro expansion is active.

### Example

```
IF @ARG(TWIDDLE) ; twiddle factor provided?
```

## CNT

Macro argument count

`@CNT()`

### Description

Returns the count of the current macro expansion arguments as an integer. A warning is issued if this function is used when no macro expansion is active.

### Example

```
ARGCNT SET @CNT()
```

## MAC

Macro definition

`@MAC(`*symbol*`)`

### Description

Returns an integer 1 if *symbol* has been defined as a macro name, 0 otherwise.

### Example

```
IF @MAC(DOMUL)     ; expand macro
```

## MXP

Macro expansion

`@MXP()`

### Description

Returns an integer 1 if the Assembler is expanding a macro, 0 otherwise.

### Example

```
IF @MXP()
```

### Assembler Mode functions

This section describes the Motorola DSP Assembler mode functions:

- `CCC` on page 32
- `CHK` on page 32
- `CTR` on page 33
- `DEF` on page 33
- `EXP` on page 34
- `INT` on page 34
- `LCV` on page 34
- `LST` on page 35
- `MSP` on page 35
- `REL` on page 36

## CCC

Cumulative cycle count

`@CCC()`

### Description

Returns the cumulative cycle count as an integer. Useful in conjunction with the CC, NOCC, and CONTCC Assembler options (see the OPT directive).

### Example

```
IF @CCC() > 200
```

## CHK

Current instruction/data checksum

`@CHK()`

### Description

Returns the current instruction/data checksum value as an integer. Useful in conjunction with the CK, NOCK, and CONTCK Assembler options (see the OPT directive).

Note that assignment of the checksum value with directives other than SET could cause phasing errors due to different generated instruction values between passes.

### Example

```
CHKSUM SET @CHK()
```

## CTR

Location counter type

`@CTR({L|R})`

### Description

If `L` is specified as the argument, returns the counter number of the load location counter. If `R` is specified, returns the counter number of the runtime location counter. The counter number is returned as an integer value.

### Example

```
CNUM = @CTR(R)      ; runtime counter number
```

## DEF

Symbol definition

`@DEF(`*symbol*`)`

### Description

Returns an integer 1 if *symbol* has been defined, 0 otherwise. The *symbol* may be any label not associated with a `MACRO` or `SECTION` directive. If *symbol* is quoted, it is looked up as a `DEFINE` symbol; if it is not quoted, it is looked up as an ordinary label.

### Example

```
IF @DEF(ANGLE) ; assemble if ANGLE defined
```

## EXP

Expression check

`@EXP(`*expression*`)`

### Description

Returns an integer 1 (memory space attribute N) if the evaluation of *expression* would not result in errors. Returns 0 if the evaluation of *expression* would cause an error. No error is the output by the Assembler if *expression* contains an error. No test is made by the Assembler for warnings. The *expression* may be relative or absolute.

### Example

```
IF @EXP(1|0) ; skip on divide by zero error
```

## INT

Integer check

`@INT(`*expression*`)`

### Description

Returns an integer 1 if *expression* has an integer result, 0 otherwise. The *expression* may be relative or absolute.

### Example

```
IF @INT(TERM)
```

## LCV

Location counter value

`@LCV({L|R}[,{L|H|`*expression*`}])`

### Description

If `L` is specified as the first argument, returns the memory space attribute and value of the load location counter. If `R` is specified, returns the memory space attribute and value of the runtime location counter. The optional second argument indicates the Low, High, or numbered counter and must be separated from the first argument by a comma. If no second argument is present, the default counter (counter 0) is assumed.

The `@LCV` function does not work correctly if used to specify the runtime counter value of a relocatable overlay. This is because the resulting value is an overlay expression, and overlay expressions may not be used to set the runtime counter for a subsequent overlay. See the ORG directive for more information.

Also, `@LCV(L,...)` does not work inside a relocatable overlay. In order to obtain the load counter value for an overlay block, origin to the load space and counter immediately before the overlay and use @LCV(L) to get the beginning load counter value for the overlay.

### Example

```
ADDR = @LCV(R)      ; save runtime address
```

## LST

LIST directive flag value

`@LST()`

### Description

Returns the value of the `LIST` directive flag as an integer. Whenever a `LIST` directive is encountered in the Assembler source, the flag is incremented; when a `NOLIST` directive is encountered, the flag is decremented.

### Example

```
DUP @CVI(@ABS(@LST())) ; list unconditionally
```

## MSP

Memory space

`@MSP(`*expression*`)`

### Description

Returns the memory space attribute of *expression* as an integer value:

| | |
|---|---|
| None | 0 |

| | |
|---|---|
| Y space | 2 |
| P space | 4 |

The *expression* may be relative or absolute.

### Example

```
MEM SET @MSP(ORIGIN)
```

## REL

Relative mode

`@REL()`

### Description

Returns an integer 1 if the Assembler is operating in relative mode, 0 otherwise.

### Example

```
IF @REL()
```

3

# Software Project Management

The CodeWarrior Motorola DSP Assembler provides several directives designed to assist in the development of large software projects. Complex software projects often are divided into smaller program units. These subprograms may be written by a team of programmers in parallel, or they may be modified portions of programs that were written for a previous development effort. The Assembler provides directives to encapsulate program units and permit the free use of symbol names within subprograms without regard to symbol names used in other programs. These encapsulated program units are called sections. Sections are also the basis for relocating blocks of code and data, so that concerns about memory placement are postponed until after the assembly process.

This chapter contains these topics:

## Using Sections

A section is bounded by a `SECTION` directive and an `ENDSEC` directive. For example:

```
SECTION sectionname [GLOBAL | STATIC | LOCAL]
.
;Section source statements
.
ENDSEC
```

All symbols that are defined within a section have the *sectionname* associated with them. This serves to protect them from like-named symbols elsewhere in the program. By default, a symbol defined inside any given section is private to that section unless the `GLOBAL` or `LOCAL` qualifiers accompany the `SECTION` directive. More information on the `GLOBAL` and `LOCAL` qualifiers can be found in Data Hiding on page 38.

Any code or data inside a section is considered an indivisible block with respect to relocation. Code or data associated with a section is independently relocatable within the memory space to which it is bound, unless the `STATIC` qualifier follows the `SECTION` directive on the instruction line. More information on the `STATIC` qualifier is available in "Relocation" on page 41.

# Data Hiding

You may use sections to "hide" data and symbols from other parts of your project. This is useful for preserving name space and improving the readability of your code. Issues surrounding data hiding include:

- Symbols on page 39
- Macros on page 40
- Nesting and Fragmentation on page 41

Symbols within a section are generally distinct from other symbols used elsewhere in the source program, even if the symbol name is the same. This is true as long as the section name associated with each symbol is unique, the symbol is not declared public (`XDEF` or `GLOBAL`), and the `GLOBAL` or `LOCAL` qualifiers are not used in the section declaration. Symbols that are defined outside of a section are considered global symbols and have no explicit section name associated with them. Global symbols may be referenced freely from inside or outside of any section, as long as the global symbol name does not conflict with another symbol by the same name in a given section. Consider Listing 3.1 on page 38:

**Listing 3.1 Data hiding example**

```
SYM1      EQU         1
SYM2      EQU         2
          SECTION     EXAMPLE
SYM1      EQU         3
          MOVE        #SYM1,R0
          MOVE        #SYM2,R1
          ENDSEC
          MOVE        #SYM1,R2
```

`SYM1` and `SYM2` are global symbols, initially defined outside of any section. Then in section `EXAMPLE` another instance of `SYM1` is defined with a different value. Because `SYM1` was redefined inside the section, the value moved to `R0` is 3. Since `SYM2` is a global symbol the value moved to `R1` is 2. The last move to `R2` is outside of any section and thus the global instance of `SYM1` is used; the value moved to `R2` is 1.

## Symbols

Symbols may be shared among sections through use of the XDEF and XREF directives. The XDEF directive instructs the Assembler that certain symbol definitions that occur within the current section are to be accessible by other sections:

XDEF *symbol,symbol,…,symbol*

The XREF directive instructs the Assembler that all references to *symbol* within the current section are references to a symbol that was declared public within another section with the XDEF directive:

XREF *symbol,symbol,…,symbol*

XDEFed symbols by default are recognized only in other sections which XREF them. They can be made fully global (recognizable by sections which do not XREF them) by use of the XR option. Alternatively, the GLOBAL directive may be used within a section to make the named symbols visible outside of the section. Both the XDEF and XREF directives must be used before the symbols to which they refer are defined or used in the section. See Listing 3.2 on page 39 for another example.

**Listing 3.2 XDEF, XREF and Sections example**

```
SYM1        EQU  1
            SECTION   SECT1
            XDEF      SYM2
SYM1        EQU       2
SYM2        EQU       3
            ENDSEC
            SECTION   SECT2
            XREF      SYM2
            MOVE      #SYM1,R0
            MOVE      #SYM2,R1
            ENDSEC
            MOVE      #SYM2,R2
```

SYM1 is first defined outside of any section. Then in section SECT1 SYM2 is declared public with an XDEF directive. SYM1 is also defined locally to section SECT1. In section SECT2, SYM2 is declared external via the XREF directive, followed by a move of SYM1 to R0. Since SYM1 was defined locally to section SECT1, the Assembler uses the global value and moves a 1 to R0. Because SYM2 was declared external in section SECT1 the value moved to R1 is 3. If SYM2 had not been XREFed in section SECT2 the value moved to R1 would have been unknown at this point. In the last instruction, it is not known what value will be moved to R2, since SYM2 was not defined outside of any section or was not declared GLOBAL within a section.

If the GLOBAL qualifier follows the *sectionname* in the SECTION directive, then all symbols defined in the section until the next ENDSEC directive are considered global. The effect is as if every symbol in the section were declared with the GLOBAL directive. This is useful when a section needs to be independently relocatable, but data hiding is not required.

If the LOCAL qualifier follows the *sectionname* in the SECTION directive, then all symbols defined in the section until the next ENDSEC directive are visible to the immediately enclosing section. The effect is as if every symbol in the section were defined within the parent section. This is useful when a section needs to be independently relocatable, but data hiding within an enclosing section is not required.

Symbols that are defined with the SET directive can be made visible with XDEF only in absolute mode, and the section name associated with the symbol is the section name of the section where the symbol was first defined. This is true even if the symbol value is changed in another section.

## Macros

The division of a program into sections controls not only labels and symbols, but also macros and DEFINE directive symbols. Macros defined within a section are private to that section and are distinct from macros defined in other sections even if they have the same macro name. Macros defined outside of sections are considered global and may be used within any section. Similarly, DEFINE directive symbols defined within a section are private to that section and
DEFINE directive symbols defined outside of any section are globally applied. There are no directives that correspond to XDEF for macros or DEFINE symbols, therefore macros and DEFINE symbols defined in a section can never be accessed globally. If you need global accessibility, define the macros and DEFINE symbols outside of any section. See Listing 3.3 on page 40 for an example.

### Listing 3.3 Define and section example

```
DEFINE    DEFVAL '1'
SECTION   SECT1
DEFINE    DEFVAL '2'
MOVE      #DEFVAL,R0
ENDSEC
MOVE      #DEFVAL,R1
```

The second definition of DEFVAL is visible only inside SECT1, so the value moved to R0 is 2. However, the second move instruction is outside the scope of SECT1 and is therefore the initial definition of DEFVAL. This means that the value 1 is moved to R1.

## Nesting and Fragmentation

Sections can be nested to any level. When the Assembler encounters a nested section, the current section is stacked and the new section is used. When the `ENDSEC` directive of the nested section is encountered, the Assembler restores the old section and uses it. The `ENDSEC` directive always applies to the most recent `SECTION` directive. Nesting sections provides a measure of scoping for symbol names, in that symbols defined within a given section are visible to other sections nested within it. For example, if section B is nested inside section A, then a symbol defined in section A can be used in section B without `XDEF`ing in section A or `XREF`ing in section B. This scoping behavior can be turned off and on with the `NONS` and `NS` options, respectively.

Sections may also be split into separate parts. That is, *sectionname* can be used multiple times with `SECTION` and `ENDSEC` directive pairs. If this occurs, then these separate (but identically named) sections can access each other's symbols freely without the use of the `XREF` and `XDEF` directives. If the `XDEF` and `XREF` directives are used within one section, they apply to all sections with the same section name. The reuse of the section name is allowed to permit the program source to be arranged in an arbitrary manner (for example, all statements that reserve X space storage locations grouped together), but retain the privacy of the symbols for each section.

## Relocation

When the Assembler operates in relative mode, sections act as the basic grouping for relocation of code and data blocks. For every section defined in the source, a set of location counters is allocated for each DSP memory space. These counters are used to maintain offsets of data and instructions relative to the beginning of the section. At link time, sections can be relocated to an absolute address, loaded in a particular order, or linked contiguously as specified by the programmer. Sections which are split into parts or among files are logically recombined so that each section can be relocated as a unit.

If the `STATIC` qualifier follows the *sectionname* in the `SECTION` directive, then all code and data defined in the section until the next `ENDSEC` directive are relocated in terms of the immediately enclosing section. The effect with respect to relocation is as if all code and data in the section were defined within the parent section. This is useful when a section needs data hiding, but independent relocation is not required.

4

# Macros

Macros make programming less repetitive. The Motorola DSP Assembler provides directives to define and use macros extensively. The topics in this chapter include:

## Macro Operations

Programming applications frequently involve the coding of a repeated pattern or group of instructions. Some patterns contain variable entries which change for each repetition of the pattern. Others are subject to conditional assembly for a given occurrence of the instruction group. In either case, macros provide a shorthand notation for handling these instruction patterns. Having determined the iterated pattern, the programmer can, within the macro, designate selected fields of any statement as variable. Thereafter, by invoking a macro, the programmer can use the entire pattern as many times as needed, substituting different parameters for the designated variable portions of the statements.

When the pattern is defined, it is given a name. This name becomes the mnemonic by which the macro is subsequently invoked (called). If the name of the macro is the same as an existing Assembler directive or mnemonic opcode, the macro replaces the directive or mnemonic opcode, and a warning is issued. The warning can be avoided by the use of the `RDIRECT` directive, which is used to remove entries from the Assembler's directive and mnemonic tables. If directives or mnemonics are removed from the Assembler's tables, then no warning is issued when the Assembler processes macros whose names are the same as the removed directive or mnemonic entries.

The macro call causes source statements to be generated. The generated statements may contain substitutable arguments. The statements produced by a macro call are relatively unrestricted as to type. They can be any processor instruction, almost any Assembler directive, or any previously-defined macro. Source statements resulting from a macro call are subject to the same conditions and restrictions that are applied to statements written by the programmer.

To invoke a macro, the macro name must appear in the operation code field of a source statement. Any arguments are placed in the operand field. By suitably selecting the arguments in relation to their use as indicated by the macro definition, the programmer causes the Assembler to produce inline coding variations of the macro definition.

The effect of a macro call is to produce inline code to perform a predefined function. The code is inserted in the normal flow of the program so that the generated instructions are executed with the rest of the program each time the macro is called.

An important feature in defining a macro is the use of macro calls within the macro definition. The Assembler processes such nested macro calls at expansion time only. The nesting of a macro definition within another definition is permitted. However, the nested macro definition is not be processed until the primary macro is expanded. The macro must be defined before its appearance in a source statement operation field.

## Macro Definition

The definition of a macro consists of three parts: the header, which assigns a name to the macro and defines the dummy arguments; the body, which consists of prototype or skeleton source statements; and the terminator. The header is the `MACRO` directive, its label, and the dummy argument list. The body contains the pattern of standard source statements. The terminator is the `ENDM` directive.

The header of a macro definition has the form:

*label* `MACRO` [*dummy argument list*][*comment*]

The required *label* is the symbol by which the macro is called. The dummy argument list has the form:

[*dumarg*[,*dumarg*,...,*dumarg*]]

The dummy arguments are symbolic names that the macro processor replaces with arguments when the macro is expanded (called). Each dummy argument must obey the same rules as global symbol names. Dummy argument names that are preceded by an underscore are not allowed. Dummy arguments are separated by commas. For example, consider the macro definition in Listing 4.1 on page 44.

**Listing 4.1  NMUL Macro Definition**

```
N_R_MUL MACRO NMUL,AVEC,BVEC,RESULT header
;RESULT(I) = AVEC(I) * BVEC(I) I=1..NMUL
;where
; NMUL = number of multiplications
; AVEC = base address of array AVEC(I)
; BVEC = base address of array BVEC(I)
; RESULT = base address of array RESULT(I)
;
  MOVE     #AVEC,R0      body
  MOVE     #BVEC,R4
  MOVE      #RESULT,R1
  MOVE      X:(R0)+,D4.S  Y:(R4)+,D7.S
  DO        #NMUL,_ENDLOOP
  FMPY.S    D4,D7,D0 X:(R0)+,D4.SY:(R4)+,D7.S
  MOVE      D0.S,X:(R1)+
  _ENDLOOP
  ENDM
;terminator
```

When a macro call is executed, the dummy arguments within the macro definition (`NMUL`,`AVEC`,`BVEC`,`RESULT` in Listing 4.1 on page 44) are replaced with the corresponding argument as defined by the macro call.

All local labels within a macro are considered distinct for the currently active level of macro expansion (unless the macro local label override is used, see below). These local labels are valid for the entire macro expansion and are not considered bounded by non-local labels. Therefore, all local labels within a macro must be unique. This mechanism allows the programmer to freely use local labels within a macro definition without regard to the number of times that the macro is expanded. Non-local labels within a macro expansion are considered to be normal labels and thus cannot occur more than once unless used with the SET directive.

When specifying a local label within the body of a macro, the programmer must be aware that the label symbol is valid for the entire body of the current level of macro expansion. It is not valid for any nested macros within the current level of expansion. The example above shows why the local label feature is useful. If the macro `N_R_MUL` were called several times, there would be several `_ENDLOOP` labels resulting from the macro expansions. This is acceptable because each `_ENDLOOP` label is considered private to a particular instance of macro expansion.

It is sometimes desirable to pass local labels as macro arguments to be used within the macro as address references (e.g. `MOVE #_LABEL,R0`). The Assembler effectively disallows this, however, since underscore label references within a macro invocation are regarded as labels local to that expansion of the macro. A macro local label override is provided which causes local symbol lookup to have normal scope rather than macro call scope. If a circumflex (^) precedes an expression containing an underscore label, then at expansion the associated term is evaluated using the normal local label list rather than the macro local label list. The operator has no effect on normal labels or outside a macro expansion.

## Macro Calls

When a macro is invoked, the statement causing the action is termed a macro call. The syntax of a macro call consists of the following fields:

[*label*] *macro name* [*arguments*][*comment*]

The argument field can have the form:

[*arg*[,*arg*,...,*arg*]]

The macro call statement is made up of three fields besides the comment field: the *label*, if any, corresponds to the value of the location counter at the start of the macro expansion; the operation field which contains the macro name; and the operand field which contains substitutable arguments. Within the operand field, each calling argument of a macro call corresponds one-to-one with a dummy argument of the macro definition. For example, the `N_R_MUL` macro defined earlier could be invoked for expansion (called) by the statement

```
N_R_MUL CNT+1,VEC1,VEC2,OUT
```

where the operand field arguments, separated by commas and taken left to right, correspond to the dummy arguments `NMUL` through `RESULT`, respectively. These arguments are then substituted in their corresponding positions of the definition to produce a sequence of instructions.

Macro arguments consist of sequences of characters separated by commas. Although these can be specified as quoted strings, to simplify coding the Assembler does not require single quotes around macro argument strings. However, if an argument has an embedded comma or space, that argument must be surrounded by single quotes ('). An argument can be declared null when calling a macro. However, it must be declared explicitly null. Null arguments can be specified in four ways: by writing the delimiting commas in succession with no intervening spaces, by terminating the argument list with a comma and omitting the rest of the argument list, by declaring the argument as a null string, or by simply omitting some or all of the arguments. A null argument causes no character to be substituted in the generated statements that reference the argument. If more arguments are supplied in the macro call than appear in the macro definition, the Assembler outputs a warning.

## Dummy Argument Operators

The Assembler macro processor provides for text substitution of arguments during macro expansion. In order to make the argument substitution facility more flexible, the Assembler also recognizes certain text operators within macro definitions which allow for transformations of the argument text. These operators can be used for text concatenation, numeric conversion, and string handling.

The dummy argument operators are:

- Concatenation \ on page 47
- Return value ? on page 47
- Return hex value % on page 48
- String operator " on page 49

## Concatenation \

Dummy arguments that are intended to be concatenated with other characters must be preceded by the backslash concatenation operator (\) to separate them from the rest of the characters. The argument may precede or follow the adjoining text, but there must be no intervening blanks between the concatenation operator and the rest of the characters. To position an argument between two alphanumeric characters, place a backslash both before and after the argument name. For example, consider the macro definition in Listing 4.2 on page 47.

**Listing 4.2  SWAP_REG Macro, Concatenation Dummy Argument**

```
SWAP_REG    MACRO     REG1,REG2 ;swap REG1,REG2 using X0 as temp
            MOVE      R\REG1,X0
            MOVE      R\REG2,R\REG1
            MOVE      X0,R\REG2
            ENDM
```

If the macro in Listing 4.2 on page 47 is called with the statement

```
SWAP_REG 0,1
```

then for the macro expansion, the macro processor would substitute the character `0` for the dummy argument `REG1` and the character `1` for the dummy argument `REG2`. The concatenation operator (`\`) indicates to the macro processor that the substitution characters for the dummy arguments are to be concatenated in both cases with the character `R`. The resulting expansion of this macro call would be:

```
MOVE R0,X0
MOVE R1,R0
MOVE X0,R1
```

## Return value ?

Another macro definition operator is the question mark (?) that returns the value of a symbol. When the macro processor encounters this operator, the `?`*symbol* sequence is converted to a character string representing the decimal value of *symbol*. For example, consider the following modification of the `SWAP_REG` macro in Listing 4.3 on page 47.

**Listing 4.3  SWAP_REG Macro, Return Value Dummy Argument**

```
SWAP_SYM    MACRO   REG1,REG2 ;swap REG1,REG2 using X0 as temp
            MOVE    R\?REG1,X0
            MOVE    R\?REG2,R\?REG1
            MOVE    X0,R\?REG2
            ENDM
```

If the source file contained the following SET statements and macro call,

```
      AREG    SET          0
      BREG    SET          1
              SWAP_SYM     AREG,BREG
```

then the sequence of events would be as follows: the macro processor would first substitute the characters AREG for each occurrence of REG1 and BREG for each occurrence of REG2. For discussion purposes (this would never appear on the source listing), the intermediate macro expansion would be:

```
MOVE R\?AREG,X0
MOVE R\?BREG,R\?AREG
MOVE X0,R\?BREG
```

The macro processor would then replace ?AREG with the character 0 and ?BREG with the character 1, since 0 is the value of the symbol AREG and 1 is the value of BREG. The resulting intermediate expansion would be:

```
MOVE R\0,X0
MOVE R\1,R\0
MOVE X0,R\1
```

Next, the macro processor would apply the concatenation operator (\), and the resulting expansion as it would appear on the source listing would be:

```
MOVE R0,X0
MOVE R1,R0
MOVE X0,R1
```

## Return hex value %

The percent sign (%) is similar to the standard return value operator except that it returns the hexadecimal value of a symbol. When the macro processor encounters this operator, the %*symbol* sequence is converted to a character string representing the hexadecimal value of the *symbol*. Consider the macro definition shown in Listing 4.4 on page 48.

**Listing 4.4 GEN_LAB Macro, Return Hex Value Dummy Argument**

```
GEN_LAB     MACRO    LAB,VAL,STMT
LAB\%VAL    STMT
            ENDM
```

This macro generates a label consisting of the concatenation of the label prefix argument and a value that is interpreted as hexadecimal. If this macro were called as follows,

```
NUM     SET         10
        GEN_LAB     HEX,NUM,'NOP'
```

the macro processor would first substitute the characters `HEX` for `LAB`, then it would replace `%VAL` with the character `A`, since `A` is the hexadecimal representation for the decimal integer 10. Next, the macro processor would apply the concatenation operator (\). Finally, the string '`NOP`' would be substituted for the `STMT` argument. The resulting expansion as it would appear in the listing file would be:

```
HEXA NOP
```

The percent sign is also the character used to indicate a binary constant. If a binary constant is required inside a macro, it may be necessary to enclose the constant in parentheses or escape the constant by following the percent sign with a backslash (\).

## String operator "

Another dummy argument operator is the double quote ("). This character is replaced with a single quote by the macro processor, but the characters following the operator are still examined for dummy argument names. The effect in the macro call is to transform any enclosed dummy arguments into literal strings. For example, consider the macro definition in Listing 4.5 on page 49.

**Listing 4.5 STR_MAC Macro, String Operator Dummy Argument**

```
STR_MAC      MACRO    STRING
             DC       "STRING"
             ENDM
```

If this macro were called with the following macro expansion line,

```
STR_MAC ABCD
```

then the resulting macro expansion would be:

```
DC 'ABCD'
```

# Directives

The CodeWarrior™ for Freescale® DSP Assembler provides directives that control the assembly of the source code and its layout.

General directives fit into the following categories:

- Assembly Control on page 51
- Symbol Definition on page 59
- Data Definition and Allocation on page 67
- Macros and Conditional Assembly on page 79
- Structured Programming on page 90

## Assembly Control

This section describes the assembly-control directives:

- COMMENT on page 51
- DEFINE on page 52
- END on page 53
- FAIL on page 54
- FORCE on page 54
- INCLUDE on page 55
- ORG on page 55
- RDIRECT on page 56
- SCSJMP on page 57
- SCSREG on page 57
- UNDEF on page 58
- WARN on page 59

### COMMENT

Start comment lines.

```
COMMENT delimiter
.
.
delimiter
```

### Remarks

The `COMMENT` directive is used to define one or more lines as comments. The first non-blank character after the `COMMENT` directive is the comment delimiter. The two delimiters are used to define the comment text. The line containing the second comment delimiter is considered the last line of the comment. The comment text can include any printable characters and the comment text is reproduced in the source listing as it appears in the source file.

A label is not allowed with this directive.

### Example

```
COMMENT      + This is a one line comment +
COMMENT      * This is a multiple line
             comment. Any number of lines
             can be placed between the
             two delimiters.
            *
```

## DEFINE

Define substitution string.

`DEFINE` *symbol* *string*

### Remarks

The `DEFINE` directive is used to define substitution strings that are used on all following source lines. All succeeding lines are searched for an occurrence of *symbol*, which are replaced by *string*. This directive is useful for providing better documentation in the source program.

The *symbol* must adhere to the restrictions for non-local labels. That is, it cannot exceed 512 characters, the first of which must be alphabetic, and the remainder of which must be either alphanumeric or the underscore(_). A warning results if a new definition of a previously-defined symbol is attempted. The Assembler output listing shows lines after the `DEFINE` directive has been applied, and therefore redefined symbols are replaced by their substitution strings (unless the `NODXL` option in effect; see the `OPT` directive).

Macros represent a special case. `DEFINE` directive translations are applied to the macro definition as it is encountered. When the macro is expanded any active `DEFINE` directive translations are again applied. `DEFINE` directive symbols that are defined within a section are only applied to that section.

A label is not allowed with this directive.

### Example

```
DEFINE       ARRAYSIZ '10*5'
DS           ARRAYSIZ
```

The above two lines would be transformed by the Assembler to the following:

```
DS           10*5
```

### See also

"UNDEF" on page 58
"SECTION" on page 62

## END

End of source program.

`END` *expression*

### Remarks

The optional `END` directive indicates that the logical end of the source program has been encountered. Any statements following the `END` directive are ignored. The optional expression in the operand field can be used to specify the starting execution address of the program. The *expression* may be absolute or relocatable, but it must have a memory space attribute of Program or None. The `END` directive cannot be used in a macro expansion.

A label is not allowed with this directive.

### Example

```
END     EGIN ; BEGIN is the starting
             ; execution address
```

## FAIL

Programmer-generated error message.

`FAIL` [{*string* | *expression*}[,...]]

### Remarks

The `FAIL` directive causes the Assembler to output an error message. The total error count is incremented as with any other error. The `FAIL` directive is normally used in conjunction with conditional assembly directives for exceptional condition checking. The assembly proceeds normally after the error has been printed. An arbitrary number of strings and expressions, in any order but separated by commas with no intervening white space, can be optionally specified to describe the nature of the generated error.

A label is not allowed with this directive.

### Example

```
FAIL      'Parameter out of range'
```

### See also

“WARN” on page 59

## FORCE

Set operand forcing mode.

`FORCE {SHORT | LONG | NONE}`

### Remarks

The `FORCE` directive causes the Assembler to force all immediate, memory, and address operands to the specified mode as if an explicit forcing operator were used. Note that if a relocatable operand value forced short is determined to be too large for the instruction word, an error occurs at link time, not during assembly. Explicit forcing operators override the effect of this directive.

A label is not allowed with this directive.

### Example

```
FORCE SHORT ; force operands short
```

## INCLUDE

Include secondary file.

`INCLUDE` *string*

### Remarks

This directive is inserted into the source program at any point where a secondary file is to be included in the source input stream. The string specifies the filename of the secondary file. The filename must be compatible with the operating system and can include a directory specification. If no extension is given for the filename, a default extension of `.asm` is supplied.

The file is searched in all the search paths as shown in the **Access Paths** panel.

A label is not allowed with this directive.

### Example

```
INCLUDE 'storage\mem.asm'
```

## ORG

Initialize memory space and location counters.

`ORG` *rms[rlc][rmp][,lms[llc][lmp]]*

### Remarks

The `ORG` directive is used to specify addresses and to indicate memory space and mapping changes. It also can designate an implicit counter mode switch in the Assembler and serve as a mechanism for initiating overlays.

A label is not allowed with this directive.

Table 5.1 on page 56 describes the `ORG` directive elements.

**Table 5.1 `ORG` Directive Elements**

| Element | Description |
|---|---|
| rms | The memory space (Y or P) that is used as the runtime memory space. |
| rlc | The runtime counter, H, L, or default (if neither H or L is specified), that is associated with the *rms* and is used as the runtime location counter. |
| rmp | Indicates the runtime physical mapping to DSP memory: I - internal, E - external,<br>R - ROM, A - port A, B - port B. If not present, no explicit mapping is done. |
| lms | The memory space (X or P) that is to be used as the load memory space. |
| llc | The load counter, H, L, or default (if neither H or L is specified), that is associated with the *lms* and is used as the load location counter. |
| lmp | Indicates the load physical mapping to DSP memory: I - internal, E - external, R - ROM,<br>A - port A, B - port B. If not present, no explicit mapping is done. |

If the last half of the operand field in an `ORG` directive dealing with the load memory space and counter is not specified, then the Assembler assumes that the load memory space and load location counter are the same as the runtime memory space and runtime location counter. In this case, object code is being assembled to be loaded into the address and memory space where it is when the program is run, and is not an overlay.

## RDIRECT

Remove directive or mnemonic from table.

`RDIRECT` *symbol1,symbol2*

### Remarks

The `RDIRECT` directive is used to remove directives from the Assembler directive and mnemonic tables. If the directive or mnemonic that has been removed is later encountered in the source file, it is assumed to be a macro. Macro definitions that have the same name as Assembler directives or mnemonics cause a warning message to be output unless the `RDIRECT` directive has been used to remove the directive or mnemonic name from the Assembler's tables.

Since the effect of this directive is global, it cannot be used in an explicitly-defined section (see `SECTION` directive). An error results if the `RDIRECT` directive is encountered in a section.

A label is not allowed with this directive.

### Example

```
RDIRECT PAGE,MOVE
```

This would cause the Assembler to remove the `PAGE` directive from the directive table and the `MOVE` mnemonic from the mnemonic table.

## SCSJMP

Set structured control statement branching mode.

`SCSJMP {SHORT | LONG | NONE}`

### Remarks

The `SCSJMP` directive is analogous to the `FORCE` directive, but it only applies to branches generated automatically by structured control statements. There is no explicit way, as with a forcing operator, to force a branch short or long when it is produced by a structured control statement. This directive causes all branches resulting from subsequent structured control statements to be forced to the specified mode.

Just like the `FORCE` pseudo-op, errors can result if a value is too large to be forced short. For relocatable code, the error may not occur until the linking phase.

### Example

```
SCSJMP SHORT
```

## SCSREG

Reassign structured control statement registers.

`SCSREG` [*srcreg* [*dstreg* , [*tmpreg* , [*extreg* , ] ] ] ]

### Remarks

The `SCSREG` directive reassigns the registers used by structured control statement (SCS) directives. It is convenient for reclaiming default SCS registers when they are needed as application operands within a structured control construct. The *srcreg* is ordinarily the source register for SCS data moves. The *dstreg* is the destination register. The *tmpreg* is a temporary register for swapping SCS operands. The *extreg* is an extra register for complex SCS operations. With no arguments `SCSREG` resets the SCS registers to their default assignments.

### Example

```
SCSREG Y0,B
```

Reassign SCS source and destination registers.

### See also

"OPT" on page 97 (MEX)
"SCSJMP" on page 57

## UNDEF

Undefine `DEFINE` symbol.

`UNDEF` [*symbol*]

### Remarks

The `UNDEF` directive causes the substitution string associated with *symbol* to be released, and *symbol* no longer represents a valid `DEFINE` substitution.

A label is not allowed with this directive.

### Example

```
DEFINE     DEBUG
.
.
UNDEF      DEBUG
```

### See also

"DEFINE" on page 52

## WARN

Programmer-generated warning.

`WARN` [{*string* | *expression*}[,{*string* | *expression*}]...]

### Remarks

The `WARN` directive causes a warning message to be output by the Assembler. The total warning count is incremented as with any other warning. The `WARN` directive is normally used in conjunction with conditional assembly directives for exceptional condition checking. The assembly proceeds normally after the warning has been printed. An arbitrary number of strings and expressions, in any order but separated by commas with no intervening white space, can be optionally specified to describe the nature of the generated warning.

A label is not allowed with this directive.

### Example

```
WARN     'invalid parameter'
```

### See also

"FAIL" on page 54

## Symbol Definition

This section describes the directives used to assign a value to a symbol:

- ENDSEC on page 60
- EQU on page 60
- GLOBAL on page 61
- LOCAL on page 62
- SECTION on page 62
- SET on page 65
- SUBROUTINE on page 65
- XREF on page 66

## ENDSEC

End section.

`ENDSEC` *expression*

### Remarks

Every `SECTION` directive must be terminated by an `ENDSEC` directive.

A label is not allowed with this directive.

### Example

```
            SECTION         COEFF
            ORG             Y:
VALUES      BSC             $100    ; Initialize to
zero
            ENDSEC
```

### See also

“SECTION” on page 62

## EQU

Equate symbol to a value.

*label*    `EQU` [{X: | P:}]*expression*

### Remarks

The `EQU` directive assigns the value and memory space attribute of *expression* to the symbol *label*. If *expression* has a memory space attribute of None, then it can be optionally preceded by any of the indicated memory space qualifiers to force a memory space attribute. An error occurs if the expression has a memory space attribute other than None and is different than the forcing memory space attribute. The optional forcing memory space attribute is useful to assign a memory space attribute to an expression that consists only of constants but is intended to refer to a fixed address in a memory space.

The `EQU` directive is one of the directives that assigns a value other than the program counter to the label. The label cannot be redefined anywhere else in the program (or section, if `SECTION` directives are being used). The *expression* may be relative or absolute, but cannot include a symbol that is not yet defined (no forward references are allowed).

### Example

```
A_D_PORT     EQU     X:$4000
```

### See also

"SET" on page 65

## GLOBAL

Global section symbol declaration.

`GLOBAL` *symbol[,symbol,…,symbol]*

### Remarks

The `GLOBAL` directive is used to specify that the list of symbols is defined within the current section, and that those definitions should be accessible by all sections. This directive is only valid if used within a program block bounded by the `SECTION` and `ENDSEC` directives. If the symbols that appear in the operand field are not defined in the section, an error is generated.

A label is not allowed with this directive.

### Example

```
SECTION     IO
GLOBAL      LOOPA ; LOOPA will be globally
                  ; accessible by other sections
.
.
ENDSEC
```

### See also

"SECTION" on page 62
"XREF" on page 66

## LOCAL

Local section symbol declaration.

```
LOCAL      symbol[,symbol,…,symbol]
```

### Remarks

The `LOCAL` directive is used to specify that the list of symbols is defined within the current section, and that those definitions are explicitly local to that section. It is useful in cases where a symbol is used as a forward reference in a nested section where the enclosing section contains a like-named symbol. This directive is only valid if used within a program block bounded by the `SECTION` and `ENDSEC` directives. The `LOCAL` directive must appear before *symbol* is defined in the section. If the symbols that appear in the operand field are not defined in the section, an error is generated.

A label is not allowed with this directive.

### Example

```
SECTION          IO
LOCAL            LOOPA ; local to this section
.
.
ENDSEC
```

### See also

“SECTION” on page 62
“XREF” on page 66

## SECTION

Start section.

```
SECTION sectionname [GLOBAL | STATIC | LOCAL]
.
.
section source statements
.
```

```
.
ENDSEC
```

### Remarks

The `SECTION` directive defines the start of a section. All symbols that are defined within a section have the *symbol* associated with them as their section name. This serves to protect them from like-named symbols elsewhere in the program. By default, a symbol defined inside any given section is private to that section unless the `GLOBAL` or `LOCAL` qualifier accompanies the `SECTION` directive.

Any code or data inside a section is considered an indivisible block with respect to relocation. Code or data associated with a section is independently relocatable within the memory space to which it is bound, unless the `STATIC` qualifier follows the `SECTION` directive on the instruction line.

Symbols within a section are generally distinct from other symbols used elsewhere in the source program, even if the symbol name is the same. This is true as long as the section name associated with each symbol is unique, the symbol is not declared public (`XDEF`/`GLOBAL`), and the `GLOBAL` or `LOCAL` qualifier is not used in the section declaration. Symbols that are defined outside of a section are considered global symbols and have no explicit section name associated with them. Global symbols may be referenced freely from inside or outside of any section, as long as the global symbol name does not conflict with another symbol by the same name in a given section.

If the `GLOBAL` qualifier follows the *sectionname* in the `SECTION` directive, then all symbols defined in the section until the next `ENDSEC` directive are considered global. The effect is as if every symbol in the section were declared with `GLOBAL`. This is useful when a section needs to be independently relocatable, but data hiding is not desired.

If the `STATIC` qualifier follows the *sectionname* in the `SECTION` directive, then all code and data defined in the section until the next `ENDSEC` directive are relocated in terms of the immediately enclosing section. The effect with respect to relocation is as if all code and data in the section were defined within the parent section. This is useful when a section needs data hiding, but independent relocation is not required.

If the `LOCAL` qualifier follows the *sectionname* in the `SECTION` directive, then all symbols defined in the section until the next `ENDSEC` directive are visible to the immediately enclosing section. The effect is as if every symbol in the section were defined within the parent section. This is useful when a section needs to be independently relocatable, but data hiding within an enclosing section is not required.

The division of a program into sections controls not only labels and symbols, but also macros and `DEFINE` directive symbols. Macros defined within a section are private to that section and are distinct from macros defined in other sections even if they have the same macro name. Macros defined outside of sections are considered global and may be used within any section. Similarly, `DEFINE` directive symbols defined within a section are private to that section and `DEFINE` directive symbols defined outside of any section are globally applied. There are no directives that correspond to `XDEF` for macros or `DEFINE` symbols, and therefore, macros and `DEFINE` symbols defined in a section can never be accessed globally. If global accessibility is desired, the macros and `DEFINE` symbols should be defined outside of any section.

Sections can be nested to any level. When the Assembler encounters a nested section, the current section is stacked and the new section is used. When the `ENDSEC` directive of the nested section is encountered, the Assembler restores the old section and uses it. The `ENDSEC` directive always applies to the most previous `SECTION` directive. Nesting sections provides a measure of scoping for symbol names, in that symbols defined within a given section are visible to other sections nested within it. For example, if section B is nested inside section A, then a symbol defined in section A can be used in section B without `XDEF`ing in section A or `XREF`ing in section B. This scoping behavior can be turned off and on with the `NONS` and `NS` options, respectively.

Sections may also be split into separate parts. That is, *sectionname* can be used multiple times with `SECTION` and `ENDSEC` directive pairs. If this occurs, then these separate (but identically named) sections can access each other's symbols freely without the use of the `XREF` and `XDEF` directives. If the `XDEF` and `XREF` directives are used within one section, they apply to all sections with the same section name. The reuse of the section name is allowed to permit the program source to be arranged in an arbitrary manner (for example, all statements that reserve X space storage locations grouped together), but retain the privacy of the symbols for each section.

When the Assembler operates in relative mode (the default), sections act as the basic grouping for relocation of code and data blocks. For every section defined in the source, a set of location counters is allocated for each DSP memory space. These counters are used to maintain offsets of data and instructions relative to the beginning of the section. At link time, sections can be relocated to an absolute address, loaded in a particular order, or linked contiguously as specified by the programmer. Sections which are split into parts or among files are logically recombined so that each section can be relocated as a unit.

In relative mode, all sections are initially relocatable. However, a section or a part of a section may be made absolute either implicitly by using the `ORG` directive or explicitly through use of the `MODE` directive.

A label is not allowed with this directive.

### Example

SECTION AUDIOFILTER

### See also

“ORG” on page 55
“GLOBAL” on page 61
“LOCAL” on page 62
“XREF” on page 66
“OPT” on page 97

## SET

Set symbol to a value.

*label* `SET` *expression*

`SET` *label* *expression*

### Remarks

The `SET` directive is used to assign the value of the expression in the operand field to the label. The `SET` directive functions somewhat like the `EQU` directive. However, labels defined via the `SET` directive can have their values redefined in another part of the program (but only through the use of another `SET` directive). The `SET` directive is useful in establishing temporary or reusable counters within macros. The expression in the operand field of a `SET` directive must be absolute and cannot include a symbol that is not yet defined (no forward references are allowed).

### Example

```
COUNT      SET      0
```

### See also

“EQU” on page 60

## SUBROUTINE

Generate debugging information for a subroutine.

`SUBROUTINE "`*function*`",` *label*`,` *size*

### Remarks

The SUBROUTINE directive causes the assembler to generate debugging information for a subroutine. The subroutine uses the specified *function*. The *label* identifies the subroutine by name. The *size* value specifies the subroutine size. This directive applies to the DSP56800 and DSP56800E processors.

### Example

```
       SUBROUTINE     "FSTART_", FSTART_, FSTARTEND-
FSTART_
_FSTART:
       jsr
_FSTARTEND:
```

## XREF

External section symbol reference.

`XREF` *symbol[,symbol,...,symbol]*

### Remarks

The `XREF` directive is used to specify that the list of symbols is referenced in the current section, but is not defined within the current section. These symbols must either have been defined outside of any section or declared as globally accessible within another section using the `XDEF` directive. If the `XREF` directive is not used to specify that a symbol is defined globally and the symbol is not defined within the current section, an error is generated, and all references within the current section to such a symbol are flagged as undefined. The `XREF` directive must appear before any reference to *symbol* in the section.

A label is not allowed with this directive.

### Example

```
SECTION     FILTER
XREF        AA,CC,DD
.
.
ENDSEC
```

### See also

"SECTION" on page 62

## Data Definition and Allocation

This section describes the data-definition and allocation directives:

### ALIGN

Align expression.

`ALIGN` *expression*

#### Remarks

The ALIGN directive sets the runtime location counter to the word amount in the *expression*. This directive applies to the DSP56800 and DSP56800E processors.

#### Example

```
         ALIGN   8   ; Align to word amount in
expression
```

```
BUFFER    dc        8
```

## BSB

Block storage bit-reverse.

[*label*]        `BSB` *expression[,expression]*

### Remarks

The `BSB` directive causes the Assembler to allocate and initialize a block of words for a reverse-carry buffer. The number of words in the block is given by the first expression, which must evaluate to an absolute integer. Each word is assigned the initial value of the second expression. If there is no second expression, an initial value of zero is assumed. If the runtime location counter is not zero, this directive first advances the runtime location counter to a base address that is a multiple of $2^K$ , where $2^K$ is greater than or equal to the value of the first expression. An error occurs if the first expression contains symbols that are not yet defined (forward references) or if the expression has a value less than or equal to zero. Also, if the first expression is not a power of two, a warning is generated. Both expressions can have any memory space attribute.

If *label* is present, it is assigned the value of the runtime location counter after a valid base address has been established.

Only one word of object code is shown on the listing, regardless of how large the first expression is. However, the runtime location counter is advanced by the number of words generated.

### Example

```
BUFFER      BSB      0.5
```

### See also

"BSC" on page 68
"BSM" on page 69
"DC" on page 71

"DWARF Symbolics" on page 78

## BSC

Block storage of constant.

[*label*]  `BSC`  *expression[,expression]*

### Remarks

The `BSC` directive causes the Assembler to allocate and initialize a block of words. The number of words in the block is given by the first expression, which must evaluate to an absolute integer. Each word is assigned the initial value of the second expression. If there is no second expression, an initial value of zero is assumed. If the first expression contains symbols that are not yet defined (forward references) or if the expression has a value less than or equal to zero, an error is generated. Both expressions can have any memory space attribute.

If *label* is present, it is assigned the value of the runtime location counter at the start of the directive processing.

Only one word of object code is shown on the listing, regardless of how large the first expression is. However, the runtime location counter is advanced by the number of words generated.

### Example

```
UNUSED      BSC       $2FFF-$2EEE,$FFFFFFFF
```

### See also

“BSM” on page 69
“BSB” on page 68
“DC” on page 71

“DWARF Symbolics” on page 78

## BSM

Block storage modulo.

[*label*]  `BSM`  *expression[,expression]*

### Remarks

The `BSM` directive causes the Assembler to allocate and initialize a block of words for a modulo buffer. The number of words in the block is given by the first expression, which must evaluate to an absolute integer. Each word is assigned the initial value of the second expression. If there is no second expression, an initial value of zero is assumed. If the runtime location counter is not zero, this directive first advances the runtime location counter to a base address that is a multiple of $2^K$, where $2^K$ is greater than or equal to the value of the first expression. An error occurs if the first expression contains symbols that are not yet defined (forward references), has a value less than or equal to zero, or falls outside the range 2 <= *expression* <= m, where m is the maximum address of the target DSP. Both expressions can have any memory space attribute.

If *label* is present, it is assigned the value of the runtime location counter after a valid base address has been established.

Only one word of object code is shown on the listing, regardless of how large the first expression is. However, the runtime location counter is advanced by the number of words generated.

### Example

```
BUFFER      BSM      $2EEE,$FFFFFFFF
```

### See also

“BSC” on page 68
“BSB” on page 68
“DC” on page 71

“DWARF Symbolics” on page 78

## BUFFER

Start buffer.

`BUFFER {M | R},`*expression*

### Remarks

The `BUFFER` directive indicates the start of a buffer of the given type. Data is allocated for the buffer until an `ENDBUF` directive is encountered. Instructions and most data definition directives may appear between the `BUFFER` and `ENDBUF` pair, although `BUFFER` directives may not be nested and certain types of directives such as `MODE`, `ORG`, `SECTION`, and other buffer allocation directives may not be used. The *expression* represents the size of the buffer. If less data is allocated than the size of the buffer, the remaining buffer locations are uninitialized. If more data is allocated than the specified size of the buffer, an error is issued.

The `BUFFER` directive sets the runtime location counter to the address of a buffer of the given type, the length of which in words is equal to the value of *expression*. The buffer type may be either Modulo or Reverse-carry. If the runtime location counter is not zero, this directive first advances the runtime location counter to a base address that is a multiple of $2^K$, where $2^K$ >= *expression*. An error is issued if there is insufficient memory remaining to establish a valid base address. Unlike other buffer allocation directives, the runtime location counter is not advanced by the value of the integer expression in the operand field; the location counter remains at the buffer base address.

The result of *expression* may have any memory space attribute but must be an absolute integer greater than zero and cannot contain any forward references (symbols that have not yet been defined). If a Modulo buffer is specified, the expression must fall within the range 2 <= *expression* <= m, where m is the maximum address of the target DSP. If a Reverse-carry buffer is designated and *expression* is not a power of two, a warning is issued.

A label is not allowed with this directive.

### Example

```
            BUFFER            M,24 ; CIRCULAR BUFFER MOD
24
M_BUF       DC                0.5,0.5,0.5,0.5
            DS                20 ; REMAINDER
UNINITIALIZED
            ENDBUF
```

### See also

“BSM” on page 69
“BSB” on page 68
“DSM” on page 75
“DSR” on page 76
“ENDBUF” on page 76

## DC

Define constant.

[*label*] `DC` *arg[,arg,...,arg]*

### Remarks

The `DC` directive allocates and initializes a word of memory for each *arg* argument. The *arg* may be a numeric constant, a single- or multiple-character string constant, a symbol, or an expression. The `DC` directive may have one or more arguments separated by commas. Multiple arguments are stored in successive address locations. If multiple arguments are present, one or more of them can be null (two adjacent commas), in which case the corresponding address location is filled with zeros.

If *label* is present, it assigns the value of the runtime location counter at the start of the directive processing.

Integer arguments are stored as is; floating-point numbers are converted to binary values. Single- and multiple-character strings are handled in the following manner:

- Single-character strings are stored in a word whose lower seven bits represent the ASCII value of the character.
- Multiple-character strings represent words whose bytes are composed of concatenated sequences of the ASCII representation of the characters in the string (unless the `NOPS` option is specified; see the `OPT` directive). If the number of characters is not an even multiple of the number of bytes per DSP word, then the last word has the remaining characters left-aligned and the rest of the word is zero-filled. If the `NOPS` option is given, each character in the string is stored in a word whose lower seven bits represent the ASCII value of the character.

### Example

```
TABLE        DC       1426,253,$2662,'ABCD'
CHARS        DC       'A','B','C','D'
```

### See also

"BSC" on page 68
"DCB" on page 72

"DWARF Symbolics" on page 78

## DCB

Define constant byte.

[*label*]     `DCB`     *arg[,arg,...,arg]*

### Remarks

The `DCB` directive allocates and initializes a byte of memory for each *arg* argument. The *arg* may be a byte integer constant, a single- or multiple-character string constant, a symbol, or a byte expression. The `DCB` directive may have one or more arguments separated by commas. Multiple arguments are stored in successive byte locations. If multiple arguments are present, one or more of them can be null (two adjacent commas), in which case the corresponding byte location is filled with zeros.

If *label* is present, it is assigned the value of the runtime location counter at the start of the directive processing.

Integer arguments are stored as is, but must be byte values (e.g. within the range 0–255); floating-point numbers are not allowed. Single- and multiple-character strings are handled in the following manner:

- Single-character strings are stored in a word whose lower seven bits represent the ASCII value of the character.
- Multiple-character strings represent words whose bytes are composed of concatenated sequences of the ASCII representation of the characters in the string (unless the `NOPS` option is specified; see the `OPT` directive). If the number of characters is not an even multiple of the number of bytes per DSP word, then the last word has the remaining characters left-aligned and the rest of the word is zero-filled. If the `NOPS` option is given, each character in the string is stored in a word whose lower seven bits represent the ASCII value of the character.

### Example

```
TABLE     DCB          'two',0,'strings',0
CHARS     DCB          'A','B','C','D'
```

### See also

[“BSC” on page 68](#)
[“DC” on page 71](#)

## DCBR

Define constant with byte-order flip.

`DCBR` *expression*

### Remarks

The DCBR directive is useful for defining byte strings with byte-order flip. The byte-order flip allows C code to properly address the byte strings. This directive applies to the DSP56800E processor.

### Example

```
Fhello     DCBR     "hello world"
```

### See also

“DWARF Symbolics” on page 78

## DS

Define storage.

*label* `DS` *expression*

### Remarks

The `DS` directive reserves a block of memory, the length of which in words is equal to the value of *expression*. This directive causes the runtime location counter to be advanced by the value of the absolute integer expression in the operand field. The *expression* can have any memory space attribute. The block of memory reserved is not initialized to any value. The expression must be an integer greater than zero and cannot contain any forward references (symbols that have not yet been defined).

### Example

```
S_BUF     DS     12      ; 12-byte buffer
```

### See also

“DSM” on page 75
“DSR” on page 76

“DWARF Symbolics” on page 78

## DSB

Define storage byte.

*label* `DSB` *expression*

### Remarks

The `DSB` directive reserves a block of memory, the length of which in bytes is equal to the value of *expression*. This directive causes the runtime location counter to be advanced by the value of the absolute integer expression or the next even number, if it is odd, in the operand field. The *expression* can have any memory space attribute. The block of memory reserved is not initialized to any value. The expression must be an integer greater than zero and cannot contain any forward references (symbols that have not yet been defined).

### Example

```
S_BUF_ODD      DSB     11      ; allocates a 12-byte
buffer
```

### See also

“DS” on page 74
“DCB” on page 72

“DWARF Symbolics” on page 78

## DSM

Define modulo storage.

*label* `DSM` *expression*

### Remarks

The DSM directive reserves a block of memory, the length of which in words is equal to the value of *expression*. If the runtime location counter is not zero, this directive first advances the runtime location counter to a base address that is a multiple of $2^{K}$ , where $2^{K}$ >= *expression*. An error is issued if there is insufficient memory remaining to establish a valid base address. Next the runtime location counter is advanced by the value of the integer expression in the operand field. *expression* can have any memory space attribute. The block of memory reserved is not initialized to any given value. The result of *expression* must be an absolute integer greater than zero and cannot contain any for-ward references (symbols that have not yet been defined). The expression also must fall within the range 2 <=*expression* <= **m**, where **m** is the maximum address of the target DSP.

### Example

```
M_BUF      DSM     24
```

### See also

“DS” on page 74
“DSR” on page 76

“DWARF Symbolics” on page 78

## DSR

Define reverse carry storage.

*label* `DSR` *expression*

### Remarks

The `DSR` directive reserves a block of memory, the length of which in words is equal to the value of *expression*. If the runtime location counter is not zero, this directive first advances the runtime location counter to a base address that is a multiple of $2^K$ , where $2^K$ >= *expression*. An error is issued if there is insufficient memory remaining to establish a valid base address. Next, the runtime location counter is advanced by the value of the integer expression in the operand field. The *expression* can have any memory space attribute. The block of reserved memory is not initialized to any given value. The result of *expression* must be an absolute integer greater than zero and cannot contain any forward references (symbols that have not yet been defined). Since the `DSR` directive is useful mainly for generating FFT buffers, if *expression* is not a power of two a warning is generated.

If *label* is present, it is assigned the value of the runtime location counter after a valid base address has been established.

### Example

```
R_BUF       DSR      8
```

### See also

“DWARF Symbolics” on page 78

## ENDBUF

End buffer.

`ENDBUF`

### Remarks

The `ENDBUF` directive is used to signify the end of a buffer block. The runtime location counter remains just beyond the end of the buffer when the `ENDBUF` directive is encountered.

A label is not allowed with this directive.

### Example

```
BUF         BUFFER R,64 ; reverse-carry buffer
            ENDBUF
```

### See also

“BUFFER” on page 70

## @HB()

Return high byte.

`@hb(`*byte_string*`)`

### Remarks

The `@hb(`*byte_string*`)` directive returns the upper byte of a word address.

This directive applies to the DSP56800E processor, supported only for operands of this form:

`X:xxxx/X:xxxxx/#xxxx`

### Example

```
move.b X:@hb(byte_string), r0 ;loads first byte of
byte_string
```

### See also

“@LB()” on page 77

## @LB()

Return low byte.

`@lb(`*byte_string*`)`

### Remarks

The `@lb(`*byte_string*`)` directive returns the lower byte of a word address.

This directive applies to the DSP56800E processor, supported only for operands of this form:

`X:xxxx/X:xxxxx/#xxxx`

### Example

```
move.b  X:@lb(byte_string), r0 ;loads first byte of
byte_string
```

### See also

"@HB()" on page 77

## DWARF Symbolics

The assembler generates DWARF symbolic information for labeled data that are defined using one of the following data definition directive types: BS, DC or DS.

The C equivalent definition for each supported data definition directive is shown in Table 5.2 on page 78.

**Table 5.2 C Definition for Supported Data Definition Directives**

| Assembler Definition | C Definition |
|---|---|
| `v: BSB n` | int[n] v = {0,...,0}; |
| `v: BSB n,m` | int[n] v = {m,...,m}; |
| `v: BSC n` | int[n] v = {0}; |
| `v: BSC n,m` | int[n] v = {m}; |
| `v: BSM n` | int[n] v = {0}; |
| `v: BSM n,m` | int[n] v = {m}; |
| `v: DC n` | int v = n; |
| `v: DC n1,n2,...,nM` | int[M] v = {n1, n2,...,nM}; |
| `v: DCBR n` | char v = n; |
| `v: DCBR n1,n2,...,nM` | char [M] v = {n1, n2,...,nM}; |
| v: DS n | int [n] v; |
| v: DSB n | char [n] v; |
| v: DSM n | int [n] v; |
| v: DSR n | int [n] v; |

**NOTE** Only one BS, DC or DS directive can be tied to a label for the symbolic information. If more than one directive is used, only the first is used and the rest are ignored. Any directives other than BS, DC, and DS are ignored.

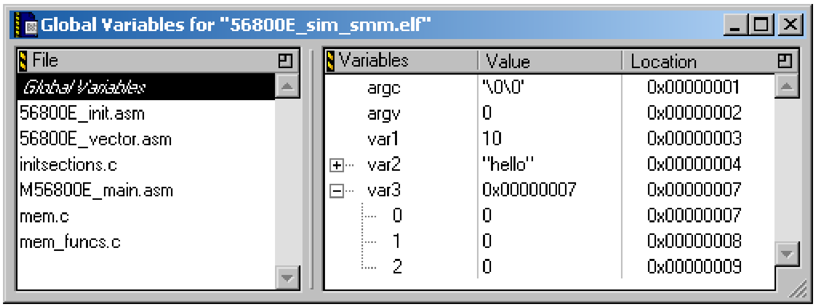
The variables may be viewed in the Global Variables window when the **Global Variables** list item are selected.

### Example

The following data definitions appear in the Global Variables window shown in Figure 5.1 on page 79:

```
var1: dc   10
var2: dcbr "hello", 0
var3: ds   3
```

**Figure 5.1 Global Variables Window**



Global Variables for "56800E_sim_smm.elf"

File
*Global Variables*
56800E_init.asm
56800E_vector.asm
initsections.c
M56800E_main.asm
mem.c
mem_funcs.c

| Variables | Value | Location |
|---|---|---|
| argc | '\0\0' | 0x00000001 |
| argv | 0 | 0x00000002 |
| var1 | 10 | 0x00000003 |
| var2 | "hello" | 0x00000004 |
| var3 | 0x00000007 | 0x00000007 |
| 0 | 0 | 0x00000007 |
| 1 | 0 | 0x00000008 |
| 2 | 0 | 0x00000009 |

## Macros and Conditional Assembly

This section describes macros and conditional-assembly directives:

- DUP on page 80
- DUPA on page 81
- DUPC on page 82
- DUPF on page 83
- ENDIF on page 84
- ENDM on page 84
- ENTRFIRQ on page 85

- ENTRXP on page 85
- EXITM on page 86
- EXITXP on page 87
- IF on page 87
- MACRO on page 89
- PMACRO on page 90

## DUP

Duplicate sequence of source lines.

```
[label]  DUP    expression
         .
         .
         ENDM
```

### Remarks

The sequence of source lines between the DUP and ENDM directives are duplicated by the number specified by the integer *expression*. The *expression* can have any memory space attribute. If the expression evaluates to a number less than or equal to 0, the sequence of lines are not included in the Assembler output. The expression result must be an absolute integer and cannot contain any forward references (symbols that have not already been defined). The DUP directive may be nested to any level.

If *label* is present, it assigns the value of the runtime location counter at the start of the DUP directive processing.

### Example

The sequence of source input statements,

```
COUNT       SET         3
            DUP         COUNT ; ASR BY COUNT
            NOP
            ENDM
```

would generate the following in the source listing:

```
COUNT       SET         3
            NOP
```

```
NOP
NOP
```

### See also

“DUPA” on page 81
“DUPC” on page 82
“DUPF” on page 83
“ENDM” on page 84
“MACRO” on page 89

## DUPA

Duplicate sequence with arguments.

[*label*] `DUPA` *dummy,arg[,arg,...,arg]*

```
.
.
ENDM
```

### Remarks

The block of source statements defined by the `DUPA` and `ENDM` directives is repeated for each argument. For each repetition, every occurrence of the dummy parameter within the block is replaced with each succeeding argument string. If the argument string is a null, then the block is repeated with each occurrence of the dummy parameter removed. If an argument includes an embedded blank or other Assembler-significant character, it must be enclosed within single quotes.

If *label* is present, it is assigned the value of the runtime location counter at the start of the `DUPA` directive processing.

### Example

If the input source file contained the following statements,

```
DUPA         VALUE,12,32,34
DC           VALUE
ENDM
```

then the assembled source listing would show

```
DC           12
DC           32
DC          34
```

### See also

“DUP” on page 80
“DUPC” on page 82
“DUPF” on page 83
“ENDM” on page 84
“MACRO” on page 89

## DUPC

Duplicate sequence with characters.

```
[label]      DUPC    dummy,string
           .
           .
            ENDM
```

### Remarks

The block of source statements defined by the `DUPC` and `ENDM` directives are repeated for each character of *string*. For each repetition, every occurrence of the *dummy* parameter within the block is replaced with each succeeding character in the string. If the string is null, then the block is skipped.

If label is present, it is assigned the value of the runtime location counter at the start of the `DUPC` directive processing.

### Example

If the input source file contained the following statements,

```
DUPC        VALUE,'123'
DC           VALUE
ENDM
```

then the assembled source listing would show:

```
DC          1
DC          2
DC          3
```

### See also

“DUP” on page 80
“DUPA” on page 81
“DUPF” on page 83
“ENDM” on page 84
“MACRO” on page 89

## DUPF

Duplicate sequence in loop.

[*label*] `DUPF` *dummy,[start],end[,increment]*

```
                .
                .
                ENDM
```

### Remarks

The block of source statements defined by the `DUPF` and `ENDM` directives are repeated in general (*end* - *start*) + 1 times when the *increment* is 1. The *start* is the starting value for the loop index; *end* represents the final value. The *increment* is the increment for the loop index; it defaults to 1 if omitted (as does the *start* value). The *dummy* parameter holds the loop index value and may be used within the body of instructions.

### Example

If the input source file contained the following statements,

```
DUPF            NUM,0,3
MOVE            #1,R\NUM
ENDM
```

then the assembled source listing would show:

```
MOVE            #1,R0
MOVE            #1,R1
MOVE            #1,R2
MOVE            #1,R3
```

### See also

"DUP" on page 80
"DUPA" on page 81
"DUPC" on page 82
"ENDM" on page 84
"MACRO" on page 89

## ENDIF

End of conditional assembly.

```
ENDIF
```

### Remarks

The `ENDIF` directive is used to signify the end of the current level of conditional assembly. Conditional assembly directives can be nested to any level, but the `ENDIF` directive always refers to the most previous `IF` directive.

### Example

```
            IF          @REL()
SAVEPC       SET          *           ; Save program
counter
            ENDIF
```

### See also

"IF" on page 87

## ENDM

End of macro definition.

```
ENDM
```

### Remarks

Every `MACRO`, `DUP`, `DUPA`, and `DUPC` directive must be terminated by an `ENDM` directive.

A label is not allowed with this directive.

### Example

```
SWAP_SYM     MACRO      REG1,REG
            MOVE       R\?REG1,D4.L
            MOVE       R\?REG2,R\?REG1
            MOVE       D4.L,R\?REG2
            ENDM
```

### See also

“DUP” on page 80
“DUPA” on page 81
“DUPC” on page 82
“DUPF” on page 83
“MACRO” on page 89

## ENTRFIRQ

Start checking for invalid P memory instructions.

```
   ENTRFIRQ      firq_handler:
```

### Remarks

The `ENTRFIRQ` directive causes the assembler to flag the first 4 to 5 instructions in a `firq` service routine, in which certain instructions are not allowed. The assembler processes these flagged instructions before the first instruction of a faster interrupt handler.

The core reference manual contains information on Fast Interrupt Processing.

This directive applies to the DSP56800E processor.

### Example

```
            ENTRFIRQ      firq_handler:
            .
            .
```

## ENTRXP

Start checking for invalid P memory instructions.

```
ENTRXP
```

### Remarks

The `ENTRXP` directive causes the assembler to verify that all instructions between the `ENTRXP` directive and the `EXITXP` directive avoid using restricted instructions not allowed for programs running from P memory space. A restricted instruction is any instruction with a dual parallel read, or any move instruction that accesses program memory.

This directive applies to the DSP56800E processor.

### Example

```
              ENTRXP
                                   ; insert here program that runs in P
memory
             EXITXP
```

### See also

“EXITXP” on page 87

## EXITM

Exit macro.

```
EXITM
```

### Remarks

The `EXITM` directive causes immediate termination of a macro expansion. It is useful when used with the conditional assembly directive `IF` to terminate macro expansion when error conditions are detected.

A label is not allowed with this directive.

### Example

```
CALC    MACRO       XVAL,YVAL
        IF          XVAL<0
        FAIL        'Macro parameter value out of range'
        EXITM       ; Exit macro
        ENDIF
```

```
          .
          .
          ENDM
```

### See also

“DUP” on page 80
“DUPA” on page 81
“DUPC” on page 82
“MACRO” on page 89

## EXITXP

Stop checking for invalid P memory instructions.

```
EXITXP
```

### Remarks

The `EXITXP` directive indicates the end of the most recent `ENTRXP` directive. The `ENTRXP` directive causes the assembler to verify that all instructions between the `ENTRXP` directive and the `EXITXP` directive avoid using restricted instructions not allowed for programs running from P memory space. A restricted instruction is any instruction with a dual parallel read, or any move instruction that accesses program memory.

This directive applies to the DSP56800E processor.

### Example

```
        ENTRXP
                      ; insert here program that runs in P
memory
        EXITXP
```

### See also

“ENTRXP” on page 85

## IF

Conditional assembler directive.

```
IF  expression
.
.
[ELSE]
.
.
ENDIF
```

## Remarks

Part of a program that is to be conditionally assembled must be bounded by an `IF`-`ENDIF` directive pair. If the optional `ELSE` directive is not present, then the source statements following the `IF` directive and up to the next `ENDIF` directive are included as part of the source file being assembled only if the *expression* has a nonzero result. If the *expression* has a value of zero, the source file is assembled as if those statements between the `IF` and the `ENDIF` directives were never encountered. If the `ELSE` directive is present and *expression* has a nonzero result, then the statements between the `IF` and `ELSE` directives are assembled, and the statements between the `ELSE` and `ENDIF` directives are skipped. Alternatively, if *expression* has a value of zero, then the statements between the `IF` and `ELSE` directives are  skipped, and the statements between the `ELSE` and `ENDIF` directives are assembled.

The *expression* must have an absolute integer result and is considered true if it has a nonzero result. The *expression* is false only if it has a result of 0. Because of the nature of the directive, *expression* must be known on pass one (no forward references allowed). `IF` directives can be nested to any level. The `ELSE` directive is always refered to the nearest previous `IF` directive as is the `ENDIF` directive.

A label is not allowed with this directive.

## Example

```
IF          @LST()>0
DUP         @LST()         ; Unwind LIST directive stack
NOLIST
ENDM
ENDIF
```

## See also

"ENDIF" on page 84

## MACRO

Macro definition.

```
label     MACRO [argumentlist]
          .
          .
          macro definition statements
          .
          .
          ENDM
```

### Remarks

The required label is the symbol by which the macro is called. If the macro is named the same as an existing Assembler directive or mnemonic, a warning is issued. This warning can be avoided with the `RDIRECT` directive.

The definition of a macro consists of three parts: the header, which assigns a name to the macro and defines the dummy arguments; the body, which consists of prototype or skeleton source statements; and the terminator. The header is the `MACRO` directive, its label, and the dummy argument list. The body contains the pattern of standard source statements. The terminator is the `ENDM` directive.

The dummy arguments are symbolic names that the macro processor replaces with arguments when the macro is expanded (called). Each dummy argument must obey the same rules as symbol names. Dummy argument names that are preceded by an underscore are not allowed. Within each of the three dummy argument fields, the dummy arguments are separated by commas. The dummy argument fields are separated by one or more blanks.

Macro definitions may be nested, but the nested macro is not defined until the primary macro is expanded.

### Example

```
;swap REG1,REG2 using X0 as temp
SWAP_SYM    MACRO   REG1,REG2
            MOVE    R\?REG1,X0
            MOVE    R\?REG2,R\?REG1
            MOVE    X0,R\?REG2
            ENDM
```

### See also

“DUP” on page 80
“DUPA” on page 81
“DUPC” on page 82
“DUPF” on page 83
“ENDM” on page 84

## PMACRO

Purge macro definition.

`PMACRO` *symbol[,symbol,...,symbol]*

### Remarks

The specified macro definition is purged from the macro table, allowing the macro table space to be reclaimed.

A label is not allowed with this directive.

### Example

```
PMACRO MAC1,MAC2
```

### See also

“MACRO” on page 89

# Structured Programming

This section describes structured-programming directives:

- .BREAK on page 91
- .CONTINUE on page 91
- .FOR and .ENDF on page 92
- .IF, .ELSE, and .ENDI on page 93
- .REPEAT and .UNTIL on page 94
- .WHILE and .ENDW on page 94

Assembly language provides an instruction set for performing certain rudimentary operations. These operations in turn may be combined into control structures such as loops (`FOR`, `REPEAT`, `WHILE`) or conditional branches (`IF-THEN`, `IF-THEN-ELSE`). The Assembler, however, accepts formal, high-level directives that specify these control structures, generating the appropriate assembly language instructions for their efficient implementation. This use of structured control statement directives improves the readability of assembly language programs, without compromising the desirable aspects of programming in an assembly language.

## .BREAK

Exit from structured loop construct.

`.BREAK` [*expression*]

### Remarks

The `.BREAK` statement causes an immediate exit from the innermost enclosing loop construct (`.WHILE`, `.REPEAT`, `.FOR`, `.LOOP`). If the optional *expression* is given, loop exit depends on the outcome of the condition.

A `.BREAK` statement does not exit an `.IF-THEN-.ELSE` construct. If a `.BREAK` is encountered with no loop statement active, a warning is issued.

The `.BREAK` statement should be used with care near `.ENDL` directives or near the end of `DO` loops. It generates a jump instruction which is illegal in those contexts. The optional *expression* is limited to condition code expressions only.

### Example

```
.WHILE x:(r1)+ <GT> #0 ;loop until zero is found
.
.
.IF <cs>
.BREAK ;causes exit from WHILE loop
.ENDI
.   ;any instructions here are skipped
.
.ENDW
;execution resumes here after .BREAK
```

## .CONTINUE

Continue next iteration of structured loop.

`.CONTINUE`

### Remarks

The `.CONTINUE` statement causes the next iteration of a looping construct (`.WHILE`, `.REPEAT`, `.FOR`, `.LOOP`) to begin. This means that the loop expression or operand comparison is performed immediately, bypassing any subsequent instructions.

If a `.CONTINUE` statement is encountered with no loop statement active, a warning is issued.

The `.CONTINUE` statement should be used with care near `.ENDL` directives or near the end of `DO` loops. It generates a jump instruction which is illegal in those contexts.

One or more `.CONTINUE` directives inside a `.LOOP` construct generates a `NOP` instruction just before the loop address.

### Example

```
.REPEAT
.
.
.IF <cs>
.CONTINUE ;causes immediate jump to .UNTIL
.ENDI
.     ;any instructions here are skipped
.
.UNTIL x:(r1)+ <EQ> #0 ;evaluation here after .CONTINUE
```

## .FOR and .ENDF

Begin for loop.

`.FOR` *op1* = *op2* {`TO` | `DOWNTO`} *op3* [`BY` *op4*] [`DO`]

*stmtlist*

`.ENDF`

### Remarks

Initialize *op1* to *op2* and perform *stmtlist* until *op1* is greater (`TO`) or less than (`DOWNTO`) *op3*. Makes use of a user-defined operand, *op1*, to serve as a loop counter. The `.FOR-TO` loop allows counting upward, while `.FOR-DOWNTO` allows counting downward. The programmer may specify an increment/decrement step size in *op4*, or elect the default step size of `#1` by omitting the `BY` clause. A `.FOR-`

`TO` loop is not executed if *op2* is greater than *op3* upon entry to the loop. Similarly, a `.FOR-DOWNTO` loop is not executed if *op2* is less than *op3*.

The *op1* must be a writable register or memory location. It is initialized at the beginning of the loop and updated with each pass through the loop. Any immediate operands must be preceded by a pound sign (#). Memory references must be preceded by a memory space qualifier (X: or P:).

The logic generated by the `.FOR` directive makes use of several DSP data registers. In fact, two data registers are used to hold the step and target values, respectively, throughout the loop; they are never reloaded by the generated code. It is recommended that these registers not be used within the body of the loop, or that they be saved and restored prior to loop evaluation.

The `DO` keyword is optional.

## .IF, .ELSE, and .ENDI

Begin if condition.

`.IF` *expression* [`THEN`]

*stmtlist*

[`.ELSE`

*stmtlist*]

`.ENDI`

### Remarks

If *expression* is true, execute *stmtlist* following `THEN` (the keyword `THEN` is optional); if *expression* is false, execute *stmtlist* following `.ELSE`, if present; otherwise, advance to the instruction following `.ENDI`.

In the case of nested `.IF-THEN-.ELSE` statements, each `.ELSE` refers to the most recent `.IF-THEN` sequence.

### Example

```
.IF <EQ> ; zero bit set?
.
.
.ENDI
```

## .REPEAT and .UNTIL

Begin repeat loop.

`.REPEAT`

*stmtlist*

`.UNTIL` *expression*

### Remarks

The *stmtlist* is executed repeatedly until *expression* is true. When *expression* becomes true, advance to the next instruction following `.UNTIL`.

The *stmtlist* is executed at least once, even if *expression* is true upon entry to the `.REPEAT` loop.

### Example

```
.REPEAT
.
.
.UNTIL x:(r1)+ <EQ> #0 ; loop until zero is found
```

## .WHILE and .ENDW

Begin while loop.

`.WHILE` *expression* [`DO`]

*stmtlist*

`.ENDW`

### Remarks

The *expression* is tested before execution of *stmtlist*. While *expression* remains true, *stmtlist* is executed repeatedly. When *expression* evaluates false, advance to the instruction following the `.ENDW` statement.

If *expression* is false upon entry to the `.WHILE` loop, *stmtlist* is not executed; execution continues after the `.ENDW` directive.

The `DO` keyword is optional.

### Example

```
.WHILE x:(r1)+ <GT> #0 ; loop until zero is found
.
.
.ENDW
```

6

# Options, Listings, and Errors

The Motorola DSP Assembler allows you to change its operating parameters with option directives. In addition, it generates lists of warnings and errors which you can control with even more directives.

You can change the assembler options within your code using the `OPT` directive. Specify your options in the first file listed in the project, in the prefix file, or have an included file that sets your options.

The topics in this chapter include:

## OPT

This directive turns assembler options on and off.

`OPT`*option[,option,...]*

### Remarks

The `OPT` directive is used to designate the Assembler options. Assembler options are given in the operand field of the source input file and are separated by commas. All options have a default condition. Some options are reset to their default condition at the end of pass one. Some are allowed to have the prefix `NO` attached to them, which then reverses their meaning.

### Example

```
OPTcc, NOW
```

## Listing format control

The options in Table 6.1 on page 98 control the format of the listing file. The parenthetical inserts specify *default* if the option is the default condition, and *reset* if the option is reset to its default state at the end of pass one of the Assembler. If the description contains **NO**, then the option may be reversed with the `NO` prefix.

**Table 6.1 Listing Format Options**

| Option | Description |
|---|---|
| `FC` | Fold trailing comments. Any trailing comments that are included in a source line are folded underneath the source line and aligned with the opcode field. Lines that start with the comment character are aligned with the label field in the source listing. The `FC` option is useful for displaying the source listing on 80-column devices. (*default* **NO**) |
| `FF` | Use form feeds for page ejects in the listing file. (*default* **NO**) |
| `FM` | Format Assembler messages so that the message text is aligned and broken at word boundaries. (*default* **NO**) |
| `PP` | Pretty print listing file. The Assembler attempts to align fields at a consistent column position without regard to source file formatting. (*default reset* **NO**) |
| `RC` | Space comments relatively in listing fields. By default, the Assembler always places comments at a consistent column position in the listing file. This option allows the comment field to float: on a line containing only a label and opcode, the comment would begin in the operand field. (*default* **NO**) |

## Reporting options

The options shown in Table 6.2 on page 98 control what is reported in the listing file.

**Table 6.2 Reporting Options**

| Option | Description |
|---|---|
| `CC` | Enable cycle counts and clear total cycle count. Cycle counts are shown on the output listing for each instruction. Cycle counts assume a full instruction fetch pipeline and no wait states. (*default* **NO**) |
| `CEX` | Print `DC` expansions. (*default* **NO**) |
| `CL` | Print the conditional assembly directives. (*default reset* **NO**) |

**Table 6.2 Reporting Options (*continued*)**

| Option | Description |
|---|---|
| `CM` | Preserve comment lines of macros when they are defined. Note that any comment line within a macro definition that starts with two consecutive semicolons (;;) is never preserved in the macro definition. (*default reset* **NO**) |
| `CONTC` | Re-enable cycle counts. Does not clear total cycle counts. The cycle count for each instruction is shown on the output listing. |
| `CRE` | Print a cross-reference table at the end of the source listing. This option, if used, must be specified before the first symbol in the source program is defined. |
| `DXL` | Expand `DEFINE` directive strings in listing. (*default reset* **NO**) |
| `HDR` | Generate listing header along with titles and subtitles. (*default reset* **NO**) |
| `IL` | Inhibit source listing. This option stops the Assembler from producing a source listing. |
| `LOC` | Include local labels in the symbol table and cross-reference listing. Local labels are not normally included in these listings. If neither the `S` nor `CRE` options are specified, then this option has no effect. The `LOC` option must be specified before the first symbol is encountered in the source file. |
| `MC` | Print macro calls. (*default reset* **NO**) |
| `MD` | Print macro definitions. (*default reset* **NO**) |
| `MEX` | Print macro expansions. (**NO**) |
| `MU` | Include a memory utilization report in the source listing. This option must appear before any code or data generation. |
| `NL` | Display conditional assembly (`IF-ELSE-ENDIF`) and section nesting levels on listing. (*default* **NO**) |
| `S` | Print a symbol table at the end of the source listing. This option has no effect if the `CRE` option is used. |
| `U` | Print the unassembled lines skipped due to failure to satisfy the condition of a conditional assembly directive. (*default* **NO**) |

## Message control

The options listed in Table 6.3 on page 100 control the types of Assembler messages that are generated.

**Table 6.3 Message Control Options**

| Option | Description |
|---|---|
| AE | Check address expressions for appropriate arithmetic operations. For example, this checks that only valid add or subtract operations are performed on address terms. (*default reset* **NO**) |
| MSW | Issue a warning on memory space incompatibilities. (*default reset* **NO**) |
| UR | Generate a warning at assembly time for each unresolved external reference. This option works only in relocatable mode. (*default* **NO**) |
| W | Print all warning messages. (*default reset* **NO**) |

## Symbol options

The options described in Table 6.4 on page 100 deal with the handling of symbols by the Assembler.

**Table 6.4 Symbol Options**

| Option | Description |
|---|---|
| CONST | EQU symbols are maintained as assembly time constants and are not be sent to the object file. (*default* **NO**) |
| DEX | Expand DEFINE symbols within quoted strings. Can also be done on a case-by-case basis using double-quoted strings. (*default* **NO**) |
| GL | Make all section symbols global. This has the same effect as declaring every section explicitly GLOBAL. This option must be given before any sections are defined explicitly in the source file. |
| GS | Make all sections global static. All section counters and attributes are associated with the GLOBAL section. This option must be given before any sections are defined explicitly in the source file. (*default reset* **NO**) |
| NS | Allow scoping of symbols within nested sections. (*default reset* **NO**) |
| SCL | Structured control statements generate non-local labels that ordinarily are not visible to the programmer. This can create problems when local labels are interspersed among structured control statements. This option causes the Assembler to maintain the current local label scope when a structured control statement label is encountered. (*default reset* **NO**) |

**Table 6.4 Symbol Options (*continued*)**

| Option | Description |
|---|---|
| `SCO` | Send structured control statement labels to object and listing files. Normally the Assembler does not externalize these labels. This option must appear before any symbol definition. |
| `SMS` | Preserve memory space in `SET` symbols. (*default reset* **NO**) |
| `SO` | Write symbol information to object file. |
| `XLL` | Write underscore local labels to object file. This is primarily used to aid debugging. This option, if used, must be specified before the first symbol in the source program is defined. |
| `XR` | Causes `XDEF`ed symbols to be recognized within other sections without being `XREF`ed. This option, if used, must be specified before the first symbol in the source program is encountered. |

## Assembler operation

The options shown in Table 6.5 on page 101 affect how the Assembler itself behaves.

**Table 6.5 Assembler Options**

| Option | Description |
|---|---|
| `AL` | Align load counter in overlay buffers. (*default reset* **NO**) |
| `DBL` | Split dual read instructions. (*default* **NO**) |
| `DLD` | Do not restrict directives in `DO` loops. The presence of some directives in `DO` loops does not make sense, including some `OPT` directive variations. This option suppresses errors on particular directives in loops. (*default* **NO**) |
| `EM` | Used when it is necessary to emulate 56100 instructions. This option must be used in order to use the following 56100 instructions in the DSP56800 part: `ASR16`, `IMAC`, `NEGW`, `TFR2`, `SUBL`, and `SWAP`. (*default* **NO**) |
| `INTR` | Perform interrupt location checks. Certain DSP instructions may not appear in the interrupt vector locations in program memory. This option enables the Assembler to check for these instructions when the program counter is within the interrupt vector bounds. (*default reset* **NO**) |
| `PS` | Pack strings in `DC` directive. Individual bytes in strings are packed into consecutive target words for the length of the string. (*default reset* **NO**) |

**Table 6.5 Assembler Options (*continued*)**

| Option | Description |
|---|---|
| `PSB` | Preserve sign bit in two's-complement negative operands. (*default reset* **NO**) |
| `RP` | Generate `NOP` instructions to accommodate pipeline delay. If an address register is loaded in one instruction, then the contents of the register are not available for use as a pointer until *after* the next instruction. Ordinarily, when the Assembler detects this condition, it issues an error message. The `RP` option causes the Assembler to output a `NOP` instruction into the output stream instead of issuing an error. (*default* **NO**) |
| `SVO` | Preserve object file on errors. Normally, any object file produced by the Assembler is deleted if errors occur during assembly. This option must be specified before any code or data is generated. |

# Index