Chapter 6
ASSEMBLER SIGNIFICANT CHARACTERS AND DIRECTIVES

6.1 INTRODUCTION

This chapter describes the directives that are recognized by the Motorola DSP Assembler. The Assembler directives are instructions to the Assembler rather than instructions to be directly translated into object code. In addition, this chapter describes special characters that are considered significant to the Assembler.

6.2 ASSEMBLER SIGNIFICANT CHARACTERS

There are several one and two character sequences that are significant to the Assembler. Some have multiple meanings depending on the context in which they are used. Special characters associated with expression evaluation are described in Chapter 3. Other Assembler-significant characters are:

;	-	Comment delimiter
;;	-	Unreported comment delimiter
****	-	Line continuation character or Macro dummy argument concatenation operator
?	-	Macro value substitution operator
%	-	Macro hex value substitution operator
^	-	Macro local label override operator
"	-	Macro string delimiter or Quoted string **DEFINE** expansion character
@	-	Function delimiter
*****	-	Location counter substitution
++	-	String concatenation operator
[ ]	-	Substring delimiter
<<	-	I/O short addressing mode force operator
<	-	Short addressing mode force operator
>	-	Long addressing mode force operator
#	-	Immediate addressing mode operator

#< - Immediate short addressing mode force operator
#> - Immediate long addressing mode force operator

6.3 ASSEMBLER DIRECTIVES

Assembler directives can be grouped by function into seven types:

1. Assembly control
2. Symbol definition
3. Data definition/storage allocation
4. Listing control and options
5. Object file control
6. Macros and conditional assembly
7. Structured programming

6.3.1 Assembly Control

The directives used for assembly control are:

COMMENT - Start comment lines
DEFINE - Define substitution string
END - End of source program
FAIL - Programmer generated error message
FORCE - Set operand forcing mode
HIMEM - Set high memory bounds
INCLUDE - Include secondary file
LOMEM - Set low memory bounds
MODE - Change relocation mode
MSG - Programmer generated message
ORG - Initialize memory space and location counters
RADIX - Change input radix for constants
RDIRECT - Remove directive or mnemonic from table
SCSJMP - Set structured control branching mode
SCSREG - Reassign structured control statement registers
UNDEF - Undefine **DEFINE** symbol
WARN - Programmer generated warning

6.3.2 Symbol Definition

The directives used to control symbol definition are:

ENDSEC - End section
EQU - Equate symbol to a value
GLOBAL - Global section symbol declaration
GSET - Set global symbol to a value
LOCAL - Local section symbol declaration
SECTION - Start section
SET - Set symbol to a value
XDEF - External section symbol definition
XREF - External section symbol reference

6.3.3 Data Definition/Storage Allocation

The directives used to control constant data definition and storage allocation are:

BADDR - Set buffer address
BSB - Block storage bit-reverse
BSC - Block storage of constant
BSM - Block storage modulo
BUFFER - Start buffer
DC - Define constant
DCB - Define constant byte
DS - Define storage
DSM - Define modulo storage
DSR - Define reverse carry storage
ENDBUF - End buffer

6.3.4 Listing Control and Options

The directives used to control the output listing are:

LIST - List the assembly
LSTCOL - Set listing field widths
NOLIST - Stop assembly listing
OPT - Assembler options
PAGE - Top of page/size page
PRCTL - Send control string to printer
STITLE - Initialize program subtitle
TABS - Set listing tab stops
TITLE - Initialize program title

6.3.5 Object File Control

The directives used for control of the object file are:

COBJ - Comment object code
IDENT - Object code identification record
SYMOBJ - Write symbol information to object file

6.3.6 Macros and Conditional Assembly

The directives used for macros and conditional assembly are:

DUP - Duplicate sequence of source lines
DUPA - Duplicate sequence with arguments
DUPC - Duplicate sequence with characters
DUPF - Duplicate sequence in loop
ENDIF - End of conditional assembly
ENDM - End of macro definition
EXITM - Exit macro
IF - Conditional assembly directive
MACLIB - Macro library
MACRO - Macro definition
PMACRO - Purge macro definition

6.3.7 Structured Programming

The directives used for structured programming are:

.BREAK - Exit from structured loop construct
.CONTINUE - Continue next iteration of structured loop
.ELSE - Perform following statements when **.IF** false
.ENDF - End of **.FOR** loop
.ENDI - End of **.IF** condition
.ENDL - End of hardware loop
.ENDW - End of **.WHILE** loop
.FOR - Begin **.FOR** loop
.IF - Begin **.IF** condition
.LOOP - Begin hardware loop
.REPEAT - Begin **.REPEAT** loop
.UNTIL - End of **.REPEAT** loop
.WHILE - Begin **.WHILE** loop

Individual descriptions of each of the Assembler special characters and directives follow. They include usage guidelines, functional descriptions, and examples. Structured programming directives are discussed separately in Chapter 7.

Some directives require a label field, while in many cases a label is optional. If the description of an Assembler directive does not indicate a mandatory or optional label field, then a label is not allowed on the same line as the directive. In general, it is disallowed to use the label field of a data directive (such as **DS**, **BSC**, or buffer directives) in an expression used to define the space being allocated. This is because in some cases the label value cannot be determined until the operand field is fully evaluated. For example:

```
BADDS    DS    BADDS
```

The line above is invalid because the label BADDS cannot reasonably be determined in this context.

;
Comment Delimiter Character

Any number of characters preceded by a semicolon (**;**), but not part of a literal string, is considered a comment. Comments are not significant to the Assembler, but they can be used to document the source program. Comments will be reproduced in the Assembler output listing. Comments are normally preserved in macro definitions, but this option can be turned off (see the **OPT** directive, this chapter).

Comments can occupy an entire line, or can be placed after the last Assembler-significant field in a source statement. A comment starting in the first column of the source file will be aligned with the label field in the listing file. Otherwise, the comment will be shifted right and aligned with the comment field in the listing file.

EXAMPLE:

```
; THIS COMMENT BEGINS IN COLUMN 1 OF THE SOURCE FILE

LOOP        JSR     COMPUTE     ; THIS IS A TRAILING COMMENT
                                ; THESE TWO COMMENTS ARE PRECEDED
                                ; BY A TAB IN THE SOURCE FILE
```

;;
Unreported Comment Delimiter Characters

Unreported comments are any number of characters preceded by two consecutive semi-colons (**;;**) that are not part of a literal string. Unreported comments are not considered significant by the Assembler, and can be included in the source statement, following the same rules as normal comments. However, unreported comments are never reproduced on the Assembler output listing, and are never saved as part of macro definitions.

EXAMPLE:

```
;; THESE LINES WILL NOT BE REPRODUCED
;; IN THE SOURCE LISTING
```

\
Line Continuation Character or
Macro Argument Concatenation Character

Line Continuation

The backslash character (****), if used as the last character on a line, indicates to the Assembler that the source statement is continued on the following line. The continuation line will be concatenated to the previous line of the source statement, and the result will be processed by the Assembler as if it were a single line source statement. The maximum source statement length (the first line and any continuation lines) is 512 characters.

EXAMPLE:

```
; THIS COMMENT \
EXTENDS OVER \
THREE LINES
```

Macro Argument Concatenation

The backslash (****) is also used to cause the concatenation of a macro dummy argument with other adjacent alphanumeric characters. For the macro processor to recognize dummy arguments, they must normally be separated from other alphanumeric characters by a non-symbol character. However, sometimes it is desirable to concatenate the argument characters with other characters. If an argument is to be concatenated in front of or behind some other symbol characters, then it must be followed by or preceded by the backslash, respectively.

EXAMPLE:

Suppose the source input file contained the following macro definition:

```
SWAP_REG   MACRO   REG1,REG2    ;swap REG1,REG2 using D4.L as temp
           MOVE    R\REG1,D4.L
           MOVE    R\REG2,R\REG1
           MOVE    D4.L,R\REG2
           ENDM
```

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. If this macro were called with the following statement,

```
SWAP_REG        0,1
```

the resulting expansion would be:

```
MOVE    R0,D4.L
MOVE    R1,R0
MOVE    D4.L,R1
```

?
Return Value of Symbol Character

The **?**<symbol> sequence, when used in macro definitions, will be replaced by an ASCII string representing the value of <symbol>. This operator may be used in association with the backslash (****) operator. The value of <symbol> must be an integer (not floating point).

EXAMPLE:

Consider the following macro definition:

```
SWAP_SYM   MACRO   REG1,REG2    ;swap REG1,REG2 using D4.L as temp
           MOVE    R\?REG1,D4.L
           MOVE    R\?REG2,R\?REG1
           MOVE    D4.L,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
```

the resulting expansion as it would appear on the source listing would be:

```
           MOVE    R0,D4.L
           MOVE    R1,R0
           MOVE    D4.L,R1
```

%
Return Hex Value of Symbol Character

The **%**<symbol> sequence, when used in macro definitions, will be replaced by an ASCII string representing the hexadecimal value of <symbol>. This operator may be used in association with the backslash (****) operator. The value of <symbol> must be an integer (not floating point).

EXAMPLE:

Consider the following macro definition:

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

If this macro were called as follows,

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

The resulting expansion as it would appear in the listing file would be:

```
HEXA        NOP
```

^

Macro Local Label Override

The circumflex (**^**), when used as a unary expression operator in a macro expansion, will cause any local labels in its associated term to be evaluated at normal scope rather than macro scope. This means that any underscore labels in the expression term following the circumflex will not be searched for in the macro local label list. The operator has no effect on normal labels or outside of a macro expansion. The circumflex operator is useful for passing local labels as macro arguments to be used as referents in the macro. Note that the circumflex is also used as the binary exclusive OR operator.

EXAMPLE:

Consider the following macro definition:

```
LOAD    MACRO   ADDR
        MOVE    P:^ADDR,R0
        ENDM
```

If this macro were called as follows,

```
_LOCAL
        LOAD    _LOCAL
```

the Assembler would ordinarily issue an error since _LOCAL is not defined within the body of the macro. With the override operator the Assembler recognizes the _LOCAL symbol outside the macro expansion and uses that value in the MOVE instruction.

"

Macro String Delimiter or Quoted String DEFINE Expansion Character

Macro String

The double quote (**"**), when used in macro definitions, is transformed by the macro processor into the string delimiter, the single quote (**'**). The macro processor examines the characters between the double quotes for any macro arguments. This mechanism allows the use of macro arguments as literal strings.

EXAMPLE:

Using the following macro definition,

```
CSTR    MACRO    STRING
        DC       "STRING"
        ENDM
```

and a macro call,

```
        CSTR     ABCD
```

the resulting macro expansion would be:

```
        DC       'ABCD'
```

Quoted String DEFINE Expansion

A sequence of characters which matches a symbol created with a **DEFINE** directive will not be expanded if the character sequence is contained within a quoted string. Assembler strings generally are enclosed in single quotes (**'**). If the string is enclosed in double

quotes (**"**) then **DEFINE** symbols will be expanded within the string. In all other respects usage of double quotes is equivalent to that of single quotes.

EXAMPLE:

Consider the source fragment below:

```
            DEFINE      LONG    'short'
STR_MAC     MACRO       STRING
            MSG         'This is a LONG STRING'
            MSG         "This is a LONG STRING"
            ENDM
```

If this macro were invoked as follows,

```
            STR_MAC     sentence
```

then the resulting expansion would be:

```
            MSG         'This is a LONG STRING'
            MSG         'This is a short sentence'
```

@
Function Delimiter

All Assembler built-in functions start with the @ symbol. See Section 3.8 for a full discussion of these functions.

EXAMPLE:

```
SVAL        EQU     @SQT(FVAL)      ; OBTAIN SQUARE ROOT
```

*****

Location Counter Substitution

When used as an operand in an expression, the asterisk represents the current integer value of the runtime location counter.

EXAMPLE:

```
            ORG     X:$100

XBASE       EQU     *+$20           ; XBASE = $120
```

++

String Concatenation Operator

Any two strings can be concatenated with the string concatenation operator (**++**). The two strings must each be enclosed by single or double quotes, and there must be no intervening blanks between the string concatenation operator and the two strings.

EXAMPLE:

'ABC'**++**'DEF' = 'ABCDEF'

[ ]
Substring Delimiter

[<string>,<offset><length>**]**

Square brackets delimit a substring operation. The <string> argument is the source string. <offset> is the substring starting position within <string>. <length> is the length of the desired substring. <string> may be any legal string combination, including another substring. An error is issued if either <offset> or <length> exceed the length of <string>.

EXAMPLE:

```
DEFINE      ID              [('DSP56000',3,5]       ; ID = '56000'
```

<<

I/O Short Addressing Mode Force Operator

Many DSP instructions allow an I/O short form of addressing. If the value of an absolute address is known to the Assembler on pass one, then the Assembler will always pick the shortest form of addressing consistent with the instruction format. If the absolute address is not known to the Assembler on pass one (that is, the address is a forward or external reference), then the Assembler will pick the long form of addressing by default. If this is not desired, then the I/O short form of addressing can be forced by preceding the absolute address by the I/O short addressing mode force operator (**<<**).

EXAMPLE:

Since the symbol IOPORT is a forward reference in the following sequence of source lines, the Assembler would pick the long absolute form of addressing by default:

```
            BTST    #4,Y:IOPORT
IOPORT      EQU     Y:$FFF3
```

Because the long absolute addressing mode would cause the instruction to be two words long instead of one word for the I/O short absolute addressing mode, it would be desirable to force the I/O short absolute addressing mode as shown below:

```
            BTST    #4,Y:<<IOPORT
IOPORT      EQU     Y:$FFF3
```

<

Short Addressing Mode Force Operator

Many DSP instructions allow a short form of addressing. If the value of an absolute address is known to the Assembler on pass one, or the **FORCE** SHORT directive is active, then the Assembler will always pick the shortest form of addressing consistent with the instruction format. If the absolute address is not known to the Assembler on pass one (that is, the address is a forward or external reference), then the Assembler will pick the long form of addressing by default. If this is not desired, then the short absolute form of addressing can be forced by preceding the absolute address by the short addressing mode force operator (**<**).

See also: **FORCE**

EXAMPLE:

Since the symbol DATAST is a forward reference in the following sequence of source lines, the Assembler would pick the long absolute form of addressing by default:

```
            MOVE    D0.L,Y:DATAST
DATAST      EQU     Y:$23
```

Because the long absolute addressing mode would cause the instruction to be two words long instead of one word for the short absolute addressing mode, it would be desirable to force the short absolute addressing mode as shown below:

```
            MOVE    D0.L,Y:<DATAST
DATAST      EQU     Y:$23
```

>

Long Addressing Mode Force Operator

Many DSP instructions allow a long form of addressing. If the value of an absolute address is known to the Assembler on pass one, then the Assembler will always pick the shortest form of addressing consistent with the instruction format, unless the **FORCE** LONG directive is active. If this is not desired, then the long absolute form of addressing can be forced by preceding the absolute address by the long addressing mode force operator (**>**).

See also: **FORCE**

EXAMPLE:

Since the symbol DATAST is a not a forward reference in the following sequence of source lines, the Assembler would pick the short absolute form of addressing:

```
DATAST      EQU     Y:$23
            MOVE    D0.L,Y:DATAST
```

If this is not desirable, then the long absolute addressing mode can be forced as shown below:

```
DATAST      EQU     Y:$23
            MOVE    D0.L,Y:>DATAST
```

#

Immediate Addressing Mode

The pound sign (#) is used to indicate to the Assembler to use the immediate addressing mode.

EXAMPLE:

```
CNST    EQU     $5
        MOVE    #CNST,D0.L
```

#<

Immediate Short Addressing Mode Force Operator

Many DSP instructions allow a short immediate form of addressing. If the immediate data is known to the Assembler on pass one (not a forward or external reference), or the **FORCE** SHORT directive is active, then the Assembler will always pick the shortest form of immediate addressing consistent with the instruction. If the immediate data is a forward or external reference, then the Assembler will pick the long form of immediate addressing by default. If this is not desired, then the short form of addressing can be forced using the immediate short addressing mode force operator (**#<**).

See also: **FORCE**

EXAMPLE:

In the following sequence of source lines, the symbol CNST is not known to the Assembler on pass one, and therefore, the Assembler would use the long immediate addressing form for the **MOVE** instruction.

```
            MOVE    #CNST,D0.L
CNST        EQU     $5
```

Because the long immediate addressing mode makes the instruction two words long instead of one word for the immediate short addressing mode, it may be desirable to force the immediate short addressing mode as shown below:

```
            MOVE    #<CNST,D0.L
CNST        EQU     $5
```

#>

Immediate Long Addressing Mode Force Operator

Many DSP instructions allow a long immediate form of addressing. If the immediate data is known to the Assembler on pass one (not a forward or external reference), then the Assembler will always pick the shortest form of immediate addressing consistent with the instruction, unless the **FORCE** LONG directive is active. If this is not desired, then the long form of addressing can be forced using the immediate long addressing mode force operator (**#>**).

See also: **FORCE**

EXAMPLE:

In the following sequence of source lines, the symbol CNST is known to the Assembler on pass one, and therefore, the Assembler would use the short immediate addressing form for the **MOVE** instruction.

```
CNST    EQU     $5
        MOVE    #CNST,D0.L
```

If this is not desirable, then the long immediate form of addressing can be forced as shown below:

```
CNST    EQU     $5
        MOVE    #>CNST,D0.L
```

BADDR
Set Buffer Address

BADDR <**M** | **R**>,<expression>

The **BADDR** 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 **M**odulo or **R**everse-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 will be 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 block of memory intended for the buffer is not initialized to any value.

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 **M**odulo 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 **R**everse-carry buffer is designated and <expression> is not a power of two a warning will be issued.

A label is not allowed with this directive.

See also: **BSM**, **BSB**, **BUFFER**, **DSM**, **DSR**

EXAMPLE:

```
            ORG     X:$100
M_BUF       BADDR   M,24              ; CIRCULAR BUFFER MOD 24
```

BSB
Block Storage Bit-Reverse

[<label>] **BSB** <expression>[,<expression>]

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 will occur if the first expression contains symbols that are not yet defined (forward references) or if the expression has a value of less than or equal to zero. Also, if the first expression is not a power of two a warning will be generated. Both expressions can have any memory space attribute.

<label>, if present, will be assigned the value of the runtime location counter after a valid base address has been established.

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

See also: **BSC**, **BSM**, **DC**

EXAMPLE:

```
BUFFER    BSB     BUFSIZ              ; INITIALIZE BUFFER TO ZEROS
```

BSC
Block Storage of Constant

[<label>] **BSC** <expression>[,<expression>]

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 of less than or equal to zero, an error will be generated. Both expressions can have any memory space attribute.

<label>, if present, will be assigned the value of the runtime location counter at the start of the directive processing.

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

See also: **BSM**, **BSB**, **DC**

EXAMPLE:

```
UNUSED      BSC     $2FFF-@LCV(R),$FFFFFFFF     ; FILL UNUSED EPROM
```

BSM
Block Storage Modulo

[<label>] **BSM** <expression>[,<expression>]

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 will occur if the first expression contains symbols that are not yet defined (forward references), has a value of 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.

<label>, if present, will be assigned the value of the runtime location counter after a valid base address has been established.

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

See also: **BSC**, **BSB**, **DC**

EXAMPLE:

```
BUFFER    BSM     BUFSIZ,$FFFFFFFF     ; INITIALIZE BUFFER TO ALL ONES
```

BUFFER
Start Buffer

BUFFER <**M** | **R**>,<expression>

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 buffer size. If less data is allocated than the size of the buffer, the remaining buffer locations will be 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 **M**odulo or **R**everse-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 will be 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 **M**odulo 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 **R**everse-carry buffer is designated and <expression> is not a power of two a warning will be issued.

A label is not allowed with this directive.

See also: **BADDR**, **BSM**, **BSB**, **DSM**, **DSR**, **ENDBUF**

EXAMPLE:

```
            ORG     X:$100
            BUFFER  M,24                ; CIRCULAR BUFFER MOD 24
M_BUF       DC      0.5,0.5,0.5,0.5
            DS      20                  ; REMAINDER UNINITIALIZED
            ENDBUF
```

COBJ
Comment Object File

COBJ <string>

The **COBJ** directive is used to place a comment in the object code file. The <string> will be put in the object file as a comment (refer to the object format description in Appendix E).

A label is not allowed with this directive.

See also: **IDENT**

EXAMPLE:

```
        COBJ    'Start of filter coefficients'
```

COMMENT
Start Comment Lines

COMMENT <delimiter>
.
.
<delimiter>

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 will be considered the last line of the comment. The comment text can include any printable characters and the comment text will be 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.
            *
```

DC
Define Constant

[<label>] **DC** <arg>[,<arg>,...,<arg>]

The **DC** directive allocates and initializes a word of memory for each <arg> argument. <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 will be filled with zeros. If the **DC** directive is used in L memory, the arguments will be evaluated and stored as long word quantities. Otherwise, an error will occur if the evaluated argument value is too large to represent in a single DSP word.

<label>, if present, will be assigned 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:

1. Single character strings are stored in a word whose lower seven bits represent the ASCII value of the character.

   EXAMPLE: 'R' = $000052

2. 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 will have the remaining characters left aligned and the rest of the word will be 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:

   'ABCD' = $414243
   $440000

See also: **BSC**, **DCB**

EXAMPLE:

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

DCB
Define Constant Byte

[<label>] **DCB** <arg>[,<arg>,...,<arg>]

The **DCB** directive allocates and initializes a byte of memory for each <arg> argument. <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 will be filled with zeros.

<label>, if present, will be 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:

1. Single character strings are stored in a word whose lower seven bits represent the ASCII value of the character.

   EXAMPLE: 'R' = $000052

2. 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 will have the remaining characters left aligned and the rest of the word will be 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:

   'AB',,'CD' = $414200
   $434400

See also: **BSC**, **DC**

EXAMPLE:

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

DEFINE
Define Substitution String

DEFINE <symbol> <string>

The **DEFINE** directive is used to define substitution strings that will be used on all following source lines. All succeeding lines will be searched for an occurrence of <symbol>, which will be replaced by <string>. This directive is useful for providing better documentation in the source program. <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 will result if a new definition of a previously defined symbol is attempted. The Assembler output listing will show lines after the **DEFINE** directive has been applied and therefore redefined symbols will be replaced by their substitution strings (unless the **NODXL** option in effect; see the **OPT** directive).

Macros represent a special case. **DEFINE** directive translations will be applied to the macro definition as it is encountered. When the macro is expanded any active **DEFINE** directive translations will again be applied.

DEFINE directive symbols that are defined within a section will only apply to that section. See the **SECTION** directive.

A label is not allowed with this directive.

See also: **UNDEF**

EXAMPLE:

If the following **DEFINE** directive occurred in the first part of the source program:

```
        DEFINE  ARRAYSIZ            '10 * SAMPLSIZ'
```

then the source line below:

```
        DS      ARRAYSIZ
```

would be transformed by the Assembler to the following:

```
        DS      10 * SAMPLSIZ
```

DS
Define Storage

[<label>] **DS** <expression>

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. <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).

<label>, if present, will be assigned the value of the runtime location counter at the start of the directive processing.

See also: **DSM**, **DSR**

EXAMPLE:

```
S_BUF    DS    12                    ; SAMPLE BUFFER
```

DSM
Define Modulo Storage

[<label>] **DSM** <expression>

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 will be 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 forward 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.

<label>, if present, will be assigned the value of the runtime location counter after a valid base address has been established.

See also: **DS**, **DSR**

EXAMPLE:

```
        ORG     X:$100
M_BUF   DSM     24                      ; CIRCULAR BUFFER MOD 24
```

DSR
Define Reverse Carry Storage

[<label>] **DSR** <expression>

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 will be 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 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 will be generated.

<label>, if present, will be assigned the value of the runtime location counter after a valid base address has been established.

See also: **DS**, **DSM**

EXAMPLE:

```
        ORG     X:$100
R_BUF   DSR     8       ; REVERSE CARRY BUFFER FOR 16 POINT FFT
```

DUP
Duplicate Sequence of Source Lines

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

The sequence of source lines between the **DUP** and **ENDM** directives will be duplicated by the number specified by the integer <expression>. <expression> can have any memory space attribute. If the expression evaluates to a number less than or equal to 0, the sequence of lines will not be 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.

<label>, if present, will be assigned the value of the runtime location counter at the start of the **DUP** directive processing.

See also: **DUPA**, **DUPC**, **DUPF**, **ENDM**, **MACRO**

EXAMPLE:

The sequence of source input statements,

```
COUNT   SET     3
        DUP     COUNT               ; ASR BY COUNT
        ASR     D0
        ENDM
```

would generate the following in the source listing:

```
COUNT   SET     3
        DUP     COUNT               ; ASR BY COUNT
        ASR     D0
        ASR     D0
        ASR     D0
        ENDM
```

Note that the lines

```
        DUP     COUNT                   ;ASR BY COUNT
        ENDM
```

will only be shown on the source listing if the **MD** option is enabled. The lines

```
        ASR     D0
        ASR     D0
        ASR     D0
```

will only be shown on the source listing if the **MEX** option is enabled.

See the **OPT** directive in this chapter for more information on the **MD** and **MEX** options.

DUPA
Duplicate Sequence With Arguments

```
[<label>]   DUPA        <dummy>,<arg>[<,<arg>,...,<arg>]
            .
            .
            ENDM
```

The block of source statements defined by the **DUPA** and **ENDM** directives will be 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 with single quotes.

<label>, if present, will be assigned the value of the runtime location counter at the start of the **DUPA** directive processing.

See also: **DUP**, **DUPC**, **DUPF**, **ENDM**, **MACRO**

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

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

Note that the lines

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

will only be shown on the source listing if the **MD** option is enabled. The lines

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

will only be shown on the source listing if the **MEX** option is enabled.

See the **OPT** directive in this chapter for more information on the **MD** and **MEX** options.

DUPC
Duplicate Sequence With Characters

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

The block of source statements defined by the **DUPC** and **ENDM** directives will be 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.

<label>, if present, will be assigned the value of the runtime location counter at the start of the **DUPC** directive processing.

See also: **DUP**, **DUPA**, **DUPF**, **ENDM**, **MACRO**

EXAMPLE:

If input source file contained the following statements,

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

then the assembled source listing would show:

```
DUPC    VALUE,'123'
DC      1
DC      2
DC      3
ENDM
```

Note that the lines

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

will only be shown on the source listing if the **MD** option is enabled. The lines

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

will only be shown on the source listing if the **MEX** option is enabled.

See the **OPT** directive in this chapter for more information on the **MD** and **MEX** options.

DUPF
Duplicate Sequence In Loop

```
[<label>]    DUPF    <dummy>,[<start>],<end>[,<increment>]
             .
             .
             ENDM
```

The block of source statements defined by the **DUPF** and **ENDM** directives will be repeated in general (<end> - <start>) + 1 times when <increment> is 1. <start> is the starting value for the loop index; <end> represents the final value. <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.

<label>, if present, will be assigned the value of the runtime location counter at the start of the **DUPF** directive processing.

See also: **DUP**, **DUPA**, **DUPC**, **ENDM**, **MACRO**

EXAMPLE:

If input source file contained the following statements,

```
DUPF    NUM,0,7
MOVE    #0,R\NUM
ENDM
```

then the assembled source listing would show:

```
DUPF    NUM,0,7
MOVE    #0,R0
MOVE    #0,R1
MOVE    #0,R2
MOVE    #0,R3
MOVE    #0,R4
MOVE    #0,R5
MOVE    #0,R6
MOVE    #0,R7
ENDM
```

Note that the lines

```
DUPF    NUM,0,7
ENDM
```

will only be shown on the source listing if the **MD** option is enabled. The lines

```
MOVE    #0,R0
MOVE    #0,R1
MOVE    #0,R2
MOVE    #0,R3
MOVE    #0,R4
MOVE    #0,R5
MOVE    #0,R6
MOVE    #0,R7
```

will only be shown on the source listing if the **MEX** option is enabled.

See the **OPT** directive in this chapter for more information on the **MD** and **MEX** options.

END
End of Source Program

END [<expression>]

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. <expression> may be absolute or relocatable, but must have a memory space attribute of **P**rogram or **N**one. The **END** directive cannot be used in a macro expansion.

A label is not allowed with this directive.

EXAMPLE:

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

ENDBUF
End Buffer

ENDBUF

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

A label is not allowed with this directive.

See also: **BUFFER**

EXAMPLE:

```
            ORG     X:$100
BUF         BUFFER  R,64        ; uninitialized reverse-carry buffer
            ENDBUF
```

ENDIF
End of Conditional Assembly

ENDIF

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.

A label is not allowed with this directive.

See also: **IF**

EXAMPLE:

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

ENDM
End of Macro Definition

ENDM

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

A label is not allowed with this directive.

See also: **DUP**, **DUPA**, **DUPC**, **MACRO**

EXAMPLE:

```
SWAP_SYM   MACRO    REG1,REG          ;swap REG1,REG2 using D4.L as temp
           MOVE     R\?REG1,D4.L
           MOVE     R\?REG2,R\?REG1
           MOVE     D4.L,R\?REG2
           ENDM
```

ENDSEC
End Section

ENDSEC

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

A label is not allowed with this directive.

See also: **SECTION**

EXAMPLE:

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

EQU
Equate Symbol to a Value

<label> **EQU** [{**X:** | **Y:** | **L:** | **P:** | **E:**}]<expression>

The **EQU** directive assigns the value and memory space attribute of <expression> to the symbol <label>. If <expression> has a memory space attribute of **N**one, then it can optionally be preceded by any of the indicated memory space qualifiers to force a memory space attribute. An error will occur if the expression has a memory space attribute other than **N**one and it 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).

See also: **SET**

EXAMPLE:

A_D_PORT **EQU** **X:**$4000

This would assign the value $4000 with a memory space attribute of **X** to the symbol A_D_PORT.

COMPUTE **EQU** @LCV(L)

@LCV(L) is used to refer to the value and memory space attribute of the load location counter. This value and memory space attribute would be assigned to the symbol COMPUTE.

EXITM
Exit Macro

EXITM

The **EXITM** directive will cause 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.

See also: **DUP**, **DUPA**, **DUPC**, **MACRO**

EXAMPLE:

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

FAIL
Programmer Generated Error

FAIL [{<str>|<exp>}[,{<str>|<exp>},...,{<str>|<exp>}]]

The **FAIL** directive will cause an error message to be output by the Assembler. The total error count will be 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 specified optionally to describe the nature of the generated error.

A label is not allowed with this directive.

See also: **MSG**, **WARN**

EXAMPLE:

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

FORCE
Set Operand Forcing Mode

FORCE {SHORT | LONG | NONE}

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 will occur at link time, not during assembly. Explicit forcing operators override the effect of this directive.

A label is not allowed with this directive.

See also: **<**, **>**, **#<**, **#>**

EXAMPLE:

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

GLOBAL
Global Section Symbol Declaration

GLOBAL <symbol>[,<symbol>,...,<symbol>]

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 will be generated.

A label is not allowed with this directive.

See also: **SECTION**, **XDEF**, **XREF**

EXAMPLE:

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

GSET
Set Global Symbol to a Value

<label> **GSET** <expression>

GSET <label> <expression>

The **GSET** directive is used to assign the value of the expression in the operand field to the label. The **GSET** directive functions somewhat like the **EQU** directive. However, labels defined via the **GSET** directive can have their values redefined in another part of the program (but only through the use of another **GSET** or **SET** directive). The **GSET** directive is useful for resetting a global **SET** symbol within a section, where the **SET** symbol would otherwise be considered local. The expression in the operand field of a **GSET** must be absolute and cannot include a symbol that is not yet defined (no forward references are allowed).

See also: **EQU**, **SET**

EXAMPLE:

```
COUNT   GSET    0               ; INITIALIZE COUNT
```

HIMEM
Set High Memory Bounds

HIMEM <mem>[<rl>]:<expression>[,...]

The **HIMEM** directive establishes an absolute high memory bound for code and data generation. <mem> corresponds to one of the DSP memory spaces (**X**, **Y**, **L**, **P**, **E**). <rl> is one of the letters **R** for runtime counter or **L** for load counter. The <expression> is an absolute integer value within the address range of the machine. If during assembly the specified location counter exceeds the value given by <expression>, a warning is issued.

A label is not allowed with this directive.

See also: **LOMEM**

EXAMPLE:

```
HIMEM    XR:$7FFF,YR:$7FFF    ; SET X/Y RUN HIGH MEM BOUNDS
```

IDENT
Object Code Identification Record

[<label>] **IDENT** <expression1>,<expression2>

The **IDENT** directive is used to create an identification record for the object module. If <label> is specified, it will be used as the module name. If <label> is not specified, then the filename of the source input file is used as the module name. <expression1> is the version number; <expression2> is the revision number. The two expressions must each evaluate to an integer result. The comment field of the **IDENT** directive will also be passed on to the object module.

See also: **COBJ**

EXAMPLE:

If the following line was included in the source file,

```
FFILTER     IDENT    1,2                    ; FIR FILTER MODULE
```

then the object module identification record would include the module name (FFILTER), the version number (1), the revision number (2), and the comment field (; FIR FILTER MODULE).

IF
Conditional Assembly Directive

```
IF          <expression>
.
.
[ELSE]      (the ELSE directive is optional)
.
.
ENDIF
```

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 will be 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 will be 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 will be assembled, and the statements between the **ELSE** and **ENDIF** directives will be skipped. Alternatively, if <expression> has a value of zero, then the statements between the **IF** and **ELSE** directives will be skipped, and the statements between the **ELSE** and **ENDIF** directives will be 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 will always refer to the nearest previous **IF** directive as will the **ENDIF** directive.

A label is not allowed with this directive.

See also: **ENDIF**

EXAMPLE:

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

INCLUDE
Include Secondary File

INCLUDE <string> | <<string>>

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 for first in the current directory, unless the <<string>> syntax is used, or in the directory specified in <string>. If the file is not found, and the **-I** option was used on the command line that invoked the Assembler, then the string specified with the **-I** option is prefixed to <string> and that directory is searched. If the <<string>> syntax is given, the file is searched for only in the directories specified with the **-I** option. Refer to Chapter 1, Running The Assembler.

A label is not allowed with this directive.

See also: **MACLIB**

EXAMPLE:

```
INCLUDE  'headers/io.asm'        ; Unix example

INCLUDE  'storage\mem.asm'       ; MS-DOS example

INCLUDE  <data.asm>              ; Do not look in current directory
```

LIST
List the Assembly

LIST

Print the listing from this point on. The **LIST** directive will not be printed, but the subsequent source lines will be output to the source listing. The default is to print the source listing. If the **IL** option has been specified, the **LIST** directive has no effect when encountered within the source program.

The **LIST** directive actually increments a counter that is checked for a positive value and is symmetrical with respect to the **NOLIST** directive. Note the following sequence:

```
; Counter value currently 1
LIST                    ; Counter value = 2
LIST                    ; Counter value = 3
NOLIST                  ; Counter value = 2
NOLIST                  ; Counter value = 1
```

The listing still would not be disabled until another **NOLIST** directive was issued.

A label is not allowed with this directive.

See also: **NOLIST**, **OPT**

EXAMPLE:

```
IF      LISTON
LIST                    ; Turn the listing back on
ENDIF
```

LOCAL
Local Section Symbol Declaration

LOCAL <symbol>[,<symbol>,...,<symbol>]

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 will be generated.

A label is not allowed with this directive.

See also: **SECTION**, **XDEF**, **XREF**

EXAMPLE:

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

LOMEM
Set Low Memory Bounds

LOMEM <mem>[<rl>]:<expression>[,...]

The **LOMEM** directive establishes an absolute low memory bound for code and data generation. <mem> corresponds to one of the DSP memory spaces (**X**, **Y**, **L**, **P**, **E**). <rl> is one of the letters **R** for runtime counter or **L** for load counter. The <expression> is an absolute integer value within the address range of the machine. If during assembly the specified location counter falls below the value given by <expression>, a warning is issued.

A label is not allowed with this directive.

See also: **HIMEM**

EXAMPLE:

```
LOMEM       XR:$100,YR:$100   ; SET X/Y RUN LOW MEM BOUNDS
```

LSTCOL
Set Listing Field Widths

LSTCOL [<labw>[,<opcw>[,<oprw>[,<opc2w>[,<opr2w>[,<xw>[,<yw>]]]]]]]

Sets the width of the output fields in the source listing. Widths are specified in terms of column positions. The starting position of any field is relative to its predecessor except for the label field, which always starts at the same position relative to page left margin, program counter value, and cycle count display. The widths may be expressed as any positive absolute integer expression. However, if the width is not adequate to accommodate the contents of a field, the text is separated from the next field by at least one space.

Any field for which the default is desired may be null. A null field can be indicated by two adjacent commas with no intervening space or by omitting any trailing fields altogether. If the **LSTCOL** directive is given with no arguments all field widths are reset to their default values.

A label is not allowed with this directive.

See also: **PAGE**

EXAMPLE:

```
        LSTCOL      40,,,,,20,20    ; Reset label, X, and Y data field widths
```

MACLIB
Macro Library

MACLIB <pathname>

This directive is used to specify the <pathname> (as defined by the operating system) of a directory that contains macro definitions. Each macro definition must be in a separate file, and the file must be named the same as the macro with the extension .ASM added. For example, BLOCKMV.ASM would be a file that contained the definition of the macro called BLOCKMV.

If the Assembler encounters a directive in the operation field that is not contained in the directive or mnemonic tables, the directory specified by <pathname> will be searched for a file of the unknown name (with the .ASM extension added). If such a file is found, the current source line will be saved, and the file will be opened for input as an **INCLUDE** file. When the end of the file is encountered, the source line is restored and processing is resumed. Because the source line is restored, the processed file must have a macro definition of the unknown directive name, or else an error will result when the source line is restored and processed. However, the processed file is not limited to macro definitions, and can include any legal source code statements.

Multiple **MACLIB** directives may be given, in which case the Assembler will search each directory in the order in which it is encountered.

A label is not allowed with this directive.

See also: **INCLUDE**

EXAMPLE:

```
MACLIB  'macros\mymacs\'        ; IBM PC example
MACLIB  'fftlib/'               ; UNIX example
```

MACRO
Macro Definition

```
<label>     MACRO  [<dummy argument list>]
            .
            .
            <macro definition statements>
            .
            .
            ENDM
```

The dummy argument list has the form:

[<dumarg>[,<dumarg>,...,<dumarg>]]

The required label is the symbol by which the macro will be called. If the macro is named the same as an existing Assembler directive or mnemonic, a warning will be 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 will replace 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 will not be defined until the primary macro is expanded.

Chapter 5 contains a complete description of macros.

See also: **DUP**, **DUPA**, **DUPC**, **DUPF**, **ENDM**

EXAMPLE:

```
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
```

MODE
Change Relocation Mode

MODE <ABS[OLUTE] | REL[ATIVE]>

Causes the Assembler to change to the designated operational mode. The **MODE** directive may be given at any time in the assembly source to alter the set of location counters used for section addressing. Code generated while in absolute mode will be placed in memory at the location determined during assembly. Relocatable code and data are based from the enclosing section start address. The **MODE** directive has no effect when the command line **-A** option is issued. See Chapter 4 for more information on modes, sections, and relocation.

A label is not allowed with this directive.

See also: **ORG**

EXAMPLE:

```
        MODE    ABS     ; Change to absolute mode
```

MSG
Programmer Generated Message

MSG [{<str>|<exp>}[,{<str>|<exp>},...,{<str>|<exp>}]]

The **MSG** directive will cause a message to be output by the Assembler. The error and warning counts will not be affected. The **MSG** directive is normally used in conjunction with conditional assembly directives for informational purposes. The assembly proceeds normally after the message has been printed. An arbitrary number of strings and expressions, in any order but separated by commas with no intervening white space, can be specified optionally to describe the nature of the message.

A label is not allowed with this directive.

See also: **FAIL**, **WARN**

EXAMPLE:

```
        MSG     'Generating sine tables'
```

NOLIST
Stop Assembly Listing

NOLIST

Do not print the listing from this point on (including the **NOLIST** directive). Subsequent source lines will not be printed.

The **NOLIST** directive actually decrements a counter that is checked for a positive value and is symmetrical with respect to the **LIST** directive. Note the following sequence:

```
; Counter value currently 1
LIST                    ; Counter value = 2
LIST                    ; Counter value = 3
NOLIST                  ; Counter value = 2
NOLIST                  ; Counter value = 1
```

The listing still would not be disabled until another **NOLIST** directive was issued.

A label is not allowed with this directive.

See also: **LIST**, **OPT**

EXAMPLE:

```
IF      LISTOFF
NOLIST                  ; Turn the listing off
ENDIF
```

OPT
Assembler Options

OPT <option>[,<option>,...,<option>] [<comment>]

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. Options also may be specified using the command line **-O** option (see Chapter 1). 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.

Options can be grouped by function into five different types:

1. Listing format control
2. Reporting options
3. Message control
4. Symbol options
5. Assembler operation

Listing Format Control

These options control the format of the listing file:

FC	- Fold trailing comments
FF	- Form feeds for page ejects
FM	- Format messages
PP	- Pretty print listing
RC	- Relative comment spacing

Reporting Options

These options control what is reported in the listing file:

CC	- Enable cycle counts
CEX	- Print **DC** expansions
CL	- Print conditional assembly directives
CM	- Preserve comment lines within macros
CONTC	- Continue cycle counts
CRE	- Print symbol cross-reference
DXL	- Expand **DEFINE** directive strings in listing
HDR	- Generate listing headers
IL	- Inhibit source listing
LOC	- Print local labels in cross-reference
MC	- Print macro calls
MD	- Print macro definitions
MEX	- Print macro expansions
MU	- Print memory utilization report
NL	- Print conditional assembly and section nesting levels
S	- Print symbol table
U	- Print skipped conditional assembly lines

Message Control

These options control the types of Assembler messages that are generated:

AE	- Check address expressions
IDW	- Warn on pipeline stalls
MSW	- Warn on memory space incompatibilities
NDE	- Warn on DALU pipeline interlocks
UR	- Flag unresolved references
W	- Display warning messages

Symbol Options

These options deal with the handling of symbols by the Assembler:

CONST	- Make **EQU** symbols assembly time constants
DEX	- Expand **DEFINE** symbols within quoted strings
GL	- Make all section symbols global
GS	- Make all sections global static
IC	- Ignore case in symbol names
NS	- Support symbol scoping in nested sections
SCL	- Scope structured control statement labels
SCO	- Structured control statement labels to listing/object file
SMS	- Preserve memory space in **SET** symbols
SO	- Write symbols to object file
XLL	- Write local labels to object file
XR	- Recognize **XDEF**ed symbols without **XREF**

Assembler Operation

Miscellaneous options having to do with internal Assembler operation:

AL	- Align load counter in overlay buffers
CK	- Enable checksumming
CONTCK	- Continue checksumming
DBL	- Split dual read instructions
DLD	- Do not restrict directives in loops
EM	- Emulate 56100 instructions on the 56800
INTR	- Perform interrupt location checks
LB	- Byte increment load counter
LBX	- Split load words into bytes
LDB	- Listing file debug
MI	- Scan **MACLIB** directories for include files
PS	- Pack strings
PSB	- Preserve sign bit in negative operands
PSM	- Programmable short addressing mode
RP	- Generate NOP to accommodate pipeline delay
RSV	- Check reserve data memory locations
SBM	- Sixteen bit mode support
SI	- Interpret short immediate as long or sign extended
SVO	- Preserve object file on errors

Following are descriptions of the individual options. 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.

A label is not allowed with this directive.

AE (default, reset) Check address expressions for appropriate arithmetic operations. For example, this will check that only valid add or subtract operations are performed on address terms.

AL (default, reset) Align load counter in overlay buffers.

CC Enable cycle counts and clear total cycle count. Cycle counts will be shown on the output listing for each instruction. Cycle counts assume a full instruction fetch pipeline and no wait states.

CEX Print **DC** expansions.

CK Enable checksumming of instruction and data values and clear cumulative checksum. The checksum value can be obtained using the **@CHK()** function (see Chapter 3).

CL (default, reset) Print the conditional assembly directives.

CM (default, reset) 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.

CONST **EQU** symbols are maintained as assembly time constants and will not be sent to the object file.

CONTC Re-enable cycle counts. Does not clear total cycle counts. The cycle count for each instruction will be shown on the output listing.

CONTCK Re-enable checksumming of instructions and data. Does not clear cumulative checksum value.

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.

DBL (DSP56800 only) Split dual read instructions.

DEX Expand **DEFINE** symbols within quoted strings. Can also be done on a case-by-case basis using double-quoted strings.

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.

DXL (default, reset) Expand **DEFINE** directive strings in listing.

EM (DSP56800 only) Used when it is necessary to emulate 56100 instructions. This option must be used in order to use the following 56100 instructions in the 56800 part: ASR16, IMAC, NEGW, TFR2, SUBL and SWAP.

FC Fold trailing comments. Any trailing comments that are included in a source line will be folded underneath the source line and aligned with the opcode field. Lines that start with the comment character will be aligned with the label field in the source listing. The **FC** option is useful for displaying the source listing on 80 column devices.

FF Use form feeds for page ejects in the listing file.

FM Format Assembler messages so that the message text is aligned and broken at word boundaries.

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 (default, reset in absolute mode) Make all sections global static. All section counters and attributes will be associated with the GLOBAL section. This option must be given before any sections are defined explicitly in the source file.

HDR (default, reset) Generate listing header along with titles and subtitles.

IC Ignore case in symbol, section, and macro names. This directive must be issued before any symbols, sections, or macros are defined.

IDW (DSP56300 only) (default, reset) Generate warning on instruction delays due to pipeline stalls.

IL Inhibit source listing. This option will stop the Assembler from producing a source listing.

INTR (default, reset in absolute mode) 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.

LB Increment load counter (if different from runtime) by number of bytes in DSP word to provide byte-wide support for overlays in bootstrap mode. This option must appear before any code or data generation.

LBX Split overlay load words into bytes and increment load counter by bytes. This option facilitates debugging of custom boot code. It must appear prior to any code or data generation.

LDB Use the listing file as the debug source file rather than the assembly language file. The **-L** command line option to generate a listing file must be specified for this option to take effect.

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** or **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 (default, reset) Print macro calls.

MD (default, reset) Print macro definitions.

MEX Print macro expansions.

MI Scan **MACLIB** directory paths for include files. The Assembler ordinarily looks for included files only in the directory specified in the **INCLUDE** directory or in the paths given by the **-I** command line option. If the **MI** option is used the Assembler will also look for included files in any designated **MACLIB** directories.

MSW (default, reset) Issue warning on memory space incompatibilities.

MU Include a memory utilization report in the source listing. This option must appear before any code or data generation.

NDE (DSP56300 only) (default, reset) This is used to check for DALU pipeline interlocks. It flags all interlocks that occur as a result of using the accumulator register as a destination in previous instructions.

NL Display conditional assembly (**IF-ELSE-ENDIF**) and section nesting levels on listing.

NOAE Do not check address expressions.

NOAL Do not align load counter in overlay buffers.

NOCC (default, reset) Disable cycle counts. Does not clear total cycle count.

NOCEX (default, reset) Do not print **DC** expansions.

NOCK (default, reset) Disable checksumming of instruction and data values.

NOCL Do not print the conditional assembly directives.

NOCM Do not preserve comment lines of macros when they are defined.

NOCONST (default, reset) **EQU** symbols are exported to the object file.

NODBL (DSP56800 only) (default, reset) Do not split dual read instructions.

NODEX (default, reset) Do not expand **DEFINE** symbols within quoted strings.

NODLD (default, reset) Restrict use of certain directives in DO loop.

NODXL Do not expand **DEFINE** directive strings in listing.

NOEM (DSP56800 only) (default, reset) Do not emulate 56100 instructions.

NOFC (default, reset) Inhibit folded comments.

NOFF (default, reset) Use multiple line feeds for page ejects in the listing file.

NOFM (default, reset) Do not format Assembler messages.

NOGS (default, reset in relative mode) Do not make all sections global static.

NOHDR Do not generate listing header. This also turns off titles and subtitles.

NOIDW (DSP56300 only) Do not generate warnings on pipeline stalls.

NOINTR (default, reset in relative mode) Do not perform interrupt location checks.

NOMC Do not print macro calls.

NOMD Do not print macro definitions.

NOMEX (default, reset) Do not print macro expansions.

NOMI (default, reset) Do not scan **MACLIB** directory paths for include files.

NOMSW Do not issue warning on memory space incompatibilities.

NONDE (DSP56300 only) Do not flag DALU pipeline interlocks.

NONL (default, reset) Do not display nesting levels on listing.

NONS Do not allow scoping of symbols within nested sections.

NOPP Do not pretty print listing file. Source lines are sent to the listing file as they are encountered in the source, with the exception that tabs are expanded to spaces and continuation lines are concatenated into a single physical line for printing.

NOPS Do not pack strings in **DC** directive. Individual bytes in strings will be stored one byte per word.

NOPSB Do not preserve sign bit in twos-complement negative operands.

NOPSM (DSP56166 only) (default, reset) Do not allow programmable short addressing.

NORC (default, reset) Do not space comments relatively.

NORP (default, reset) Do not generate instructions to accommodate pipeline delay.

NORSV (DSP96000 only) (default, reset) Do not perform reserve memory checks.

NOSCL Do not maintain the current local label scope when a structured control statement label is encountered.

NOSI (DSP56000 only) (default, reset) Interpret an eight-bit short immediate value moved to a fractional register as a short unless forced long.

(DSP56100 only) (default, reset) Do not interpret eighth bit of short immediate value as implied sign extension.

NOSMS Do not preserve memory space in **SET** symbols.

NOU (default, reset) Do not print the lines excluded from the assembly due to a conditional assembly directive.

NOUR (default, reset) Do not flag unresolved external references.

NOW Do not print warning messages.

NS (default, reset) Allow scoping of symbols within nested sections.

PP (default, reset) Pretty print listing file. The Assembler attempts to align fields at a consistent column position without regard to source file formatting.

PS (default, reset) Pack strings in **DC** directive. Individual bytes in strings will be packed into consecutive target words for the length of the string.

PSB (default, reset) Preserve sign bit in twos-complement negative operands.

PSM (DSP56100 only) Allow programmable short addressing, disabling short and I/O short address checking.

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.

RP Generate NOP instructions to accommodate pipeline delay. If an address register is loaded in one instruction then the contents of the register is 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 will cause the Assembler to output a NOP instruction into the output stream instead of issuing an error.

RSV (DSP96000 only) Perform location counter checks to insure code/data is not located in DSP96000 reserve data memory locations. The Assembler will issue a warning if the program counter value falls within the reserved range.

S Print symbol table at the end of the source listing. This option has no effect if the **CRE** option is used.

SBM (DSP56300 only) Supports 16 bit mode operation for the 56300 when used in such a mode. This option ensures that in evaluations of fractional values the

upper 16 bits are considered rather than the lower 16 bits. Not using this option does not preclude the use of the 16 bit mode in the 56300.

SCL (default, reset) 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.

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.

SI (DSP56000 only) Interpret an eight-bit short immediate value moved to a fractional register as a long unless forced short.

(DSP56100 only) Interpret eighth bit of short immediate as implied sign extension.

SMS (default, reset) Preserve memory space in **SET** symbols.

SO Write symbol information to object file. This option is recognized but performs no operation in COFF Assemblers.

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 given before any code or data is generated.

U Print the unassembled lines skipped due to failure to satisfy the condition of a conditional assembly directive.

UR Generate a warning at assembly time for each unresolved external reference. This option works only in relocatable mode.

W (default, reset) Print all warning messages.

WEX Add warning count to exit status. Ordinarily the Assembler exits with a count of errors. This option causes the count of warnings to be added to the error count.

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.

EXAMPLE:

```
        OPT     CEX,MEX         ; Turn on DC and macro expansions
        OPT     CRE,MU          ; Cross reference, memory utilization
```

ORG
Initialize Memory Space and Location Counters

ORG <rms>[<rlc>][<rmp>]:[<exp1>][,<lms>[<llc>][<lmp>]:[<exp2>]]

ORG
<rms>[<rmp>][(<rce>)]:[<exp1>][,<lms>[<lmp>][(<lce>)]:[<exp2>]]

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 serves as a mechanism for initiating overlays.

A label is not allowed with this directive.

<rms>

Which memory space (X, Y, L, P, or E) will be used as the runtime memory space. If the memory space is **L**, any allocated datum with a value greater than the target word size will be extended to two words; otherwise, it is truncated. If the memory space is **E**, then depending on the memory space qualifier, any generated words will be split into bytes, one byte per word, or a 16/8-bit combination.

<rlc>

Which runtime counter H, L, or default (if neither H or L is specified), that is associated with the <rms> will be used as the runtime location counter.

<rmp>

Indicates the runtime physical mapping to DSP memory: I - internal, E - external, R - ROM, A - port A, B - port B. If not present, no explicit mapping is done.

<rce>

Non-negative absolute integer expression representing the counter number to be used as the runtime location counter. Must be enclosed in parentheses. Should not exceed the value 65535.

<exp1>

Initial value to assign to the runtime counter used as the <rlc>. If <exp1> is a relative expression the Assembler uses the relative location counter. If <exp1> is an absolute expression the Assembler uses the absolute location counter. If <exp1> is not specified, then the last value and mode that the counter had will be used.

<lms>

Which memory space (X, Y, L, P, or E) will be used as the load memory space. If the memory space is **L**, any allocated datum with a value greater than the target word size will be extended to two words; otherwise, it is truncated. If the memory space is **E**, then depending on the memory space qualifier, any generated words will be split into bytes, one byte per word, or a 16/8-bit combination.

<llc>

Which load counter, H, L, or default (if neither H or L is specified), that is associated with the <lms> will be used as the load location counter.

<lmp>

Indicates the load physical mapping to DSP memory: I - internal, E - external, R - ROM, A - port A, B - port B. If not present, no explicit mapping is done.

<lce>

Non-negative absolute integer expression representing the counter number to be used as the load location counter. Must be enclosed in parentheses. Should not exceed the value 65535.

<exp2>

Initial value to assign to the load counter used as the <llc>. If <exp2> is a relative expression the Assembler uses the relative location counter. If <exp2> is an absolute expression the Assembler uses the absolute location counter. If <exp2> is not specified, then the last value and mode that the counter had will be used.

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 will assume 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 will be when the program is run, and is not an overlay.

If the load memory space and counter are given in the operand field, then the Assembler always generates code for an overlay. Whether the overlay is absolute or relocatable depends upon the current operating mode of the Assembler and whether the load counter value is an absolute or relative expression. If the Assembler is running in absolute mode, or if the load counter expression is absolute, then the overlay is absolute. If the Assembler is in relative mode and the load counter expression is relative, the overlay is relocatable. Runtime relocatable overlay code is addressed relative to the location given in the runtime

location counter expression. This expression, if relative, may not refer to another overlay block.

See also: **MODE**

EXAMPLES:

ORG P:$1000

Sets the runtime memory space to P. Selects the default runtime counter (counter 0) associated with P space to use as the runtime location counter and initializes it to $1000. The load memory space is implied to be P, and the load location counter is assumed to be the same as the runtime location counter.

ORG PHE:

Sets the runtime memory space to P. Selects the H load counter (counter 2) associated with P space to use as the runtime location counter. The H counter will not be initialized, and its last value will be used. Code generated hereafter will be mapped to external (E) memory. The load memory space is implied to be P, and the load location counter is assumed to be the same as the runtime location counter.

ORG PI:OVL1,Y:

Indicates code will be generated for an overlay. The runtime memory space is P, and the default counter is used as the runtime location counter. It will be reset to the value of OVL1. If the Assembler is in absolute mode via the **-A** command line option then OVL1 must be an absolute expression. If OVL1 is an absolute expression the Assembler uses the absolute runtime location counter. If OVL1 is a relocatable value the Assembler uses the relative runtime location counter. In this case OVL1 must not itself be an overlay symbol (e.g. defined within an overlay block). The load memory space is Y. Since neither H, L, nor any counter expression was specified as the load counter, the default load counter (counter 0) will be used as the load location counter. The counter value and mode will be whatever it was the last time it was referenced.

ORG XL:,E8:

Sets the runtime memory space to X. Selects the L counter (counter 1) associated with X space to use as the runtime location counter. The L counter will not be initialized, and its last value will be used. The load memory space is set to E, and the qualifier 8 indicates a bytewise RAM configuration. Instructions and data will be generated eight bits per output word with byte-oriented load addresses. The default load counter will be used and there is no explicit load origin.

ORG P(5):,Y:$8000

Indicates code will be generated for an absolute overlay. The runtime memory space is P, and the counter used as the runtime location counter is counter 5. It will not be initialized, and the last previous value of counter 5 will be used. The load memory space is Y. Since neither H, L, nor any counter expression was specified as the load counter, the default load counter (counter 0) will be used as the load location counter. The default load counter will be initialized to $8000.

PAGE
Top of Page/Size Page

PAGE [<exp1>[,<exp2>...,<exp5>]]

The **PAGE** directive has two forms:

1. If no arguments are supplied, then the Assembler will advance the listing to the top of the next page. In this case, the **PAGE** directive will not be output.

2. The **PAGE** directive with arguments can be used to specify the printed format of the output listing. Arguments may be any positive absolute integer expression. The arguments in the operand field (as explained below) are separated by commas. Any argument can be left as the default or last set value by omitting the argument and using two adjacent commas. The **PAGE** directive with arguments will not cause a page eject and will be printed in the source listing.

A label is not allowed with this directive.

The arguments in order are:

PAGE_WIDTH <exp1>

Page width in terms of number of output columns per line (default 80, min 1, max 255).

PAGE_LENGTH <exp2>

Page length in terms of total number of lines per page (default 66, min 10, max 255). As a special case a page length of 0 (zero) turns off all headers, titles, subtitles, and page breaks.

BLANK_TOP <exp3>

Blank lines at top of page. (default 0, min 0, max see below).

BLANK_BOTTOM <exp4>

Blank lines at bottom of page. (default 0, min 0, max see below).

BLANK_LEFT <exp5>

Blank left margin. Number of blank columns at the left of the page. (default 0, min 0, max see below).

The following relationships must be maintained:

BLANK_TOP + BLANK_BOTTOM <= PAGE_LENGTH - 10

BLANK_LEFT < PAGE_WIDTH

See also: **LSTCOL**

EXAMPLE:

```
        PAGE    132,,3,3        ; Set width to132, 3 line top/bottom margins
        PAGE                    ; Page eject
```

PMACRO
Purge Macro Definition

PMACRO <symbol>[,<symbol>,...,<symbol>]

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

A label is not allowed with this directive.

See also: **MACRO**

EXAMPLE:

PMACRO MAC1,MAC2

This statement would cause the macros named MAC1 and MAC2 to be purged.

PRCTL
Send Control String to Printer

PRCTL <exp>I<string>,...,<exp>I<string>

PRCTL simply concatenates its arguments and ships them to the listing file (the directive line itself is not printed unless there is an error). <exp> is a byte expression and <string> is an Assembler string. A byte expression would be used to encode non-printing control characters, such as ESC. The string may be of arbitrary length, up to the maximum Assembler-defined limits.

PRCTL may appear anywhere in the source file and the control string will be output at the corresponding place in the listing file. However, if a **PRCTL** directive is the last line in the last input file to be processed, the Assembler insures that all error summaries, symbol tables, and cross-references have been printed before sending out the control string. This is so a **PRCTL** directive can be used to restore a printer to a previous mode after printing is done. Similarly, if the **PRCTL** directive appears as the first line in the first input file, the control string will be output before page headings or titles.

The **PRCTL** directive only works if the **-L** command line option is given; otherwise it is ignored. See Chapter 1 for more information on the **-L** option.

A label is not allowed with this directive.

EXAMPLE:

```
        PRCTL   $1B,'E'                 ; Reset HP LaserJet printer
```

RADIX
Change Input Radix for Constants

RADIX <expression>

Changes the input base of constants to the result of <expression>. The absolute integer expression must evaluate to one of the legal constant bases (2, 10, or 16). The default radix is 10. The **RADIX** directive allows the programmer to specify constants in a preferred radix without a leading radix indicator. The radix prefix for base 10 numbers is the grave accent (`). Note that if a constant is used to alter the radix, it must be in the appropriate input base at the time the **RADIX** directive is encountered.

A label is not allowed with this directive.

EXAMPLE:

```
_RAD10      DC      10          ; Evaluates to hex A
            RADIX   2
_RAD2       DC      10          ; Evaluates to hex 2
            RADIX   `16
_RAD16      DC      10          ; Evaluates to hex 10
            RADIX   3           ; Bad radix expression
```

RDIRECT
Remove Directive or Mnemonic from Table

RDIRECT <direc>[,<direc>,...,<direc>]

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 will be assumed to be a macro. Macro definitions that have the same name as Assembler directives or mnemonics will 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. Additionally, if a macro is defined through the **MACLIB** directive which has the same name as an existing directive or opcode, it will not automatically replace that directive or opcode as previously described. In this case, the **RDIRECT** directive must be used to force the replacement.

Since the effect of this directive is global, it cannot be used in an explicitly-defined section (see **SECTION** directive). An error will result 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}

The **SCSJMP** directive is analogous to the **FORCE** directive, but it only applies to branches generated automatically by structured control statements (see Chapter 7). 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 will cause 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.

See also: **FORCE**, **SCSREG**

A label is not allowed with this directive.

EXAMPLE:

```
        SCSJMP      SHORT           ; force all subsequent SCS jumps short
```

SCSREG
Reassign Structured Control Statement Registers

SCSREG [<srcreg>[,<dstreg>[,<tmpreg>[,<extreg>]]]]

The **SCSREG** directive reassigns the registers used by structured control statement (SCS) directives (see Chapter 7). It is convenient for reclaiming default SCS registers when they are needed as application operands within a structured control construct. <srcreg> is ordinarily the source register for SCS data moves. <dstreg> is the destination register. <tmpreg> is a temporary register for swapping SCS operands. <extreg> is an extra register for complex SCS operations. With no arguments **SCSREG** resets the SCS registers to their default assignments.

The **SCSREG** directive should be used judiciously to avoid register context errors during SCS expansion. Source and destination registers may not necessarily be used strictly as source and destination operands. The Assembler does no checking of reassigned registers beyond validity for the target processor. Errors can result when a structured control statement is expanded and an improper register reassignment has occurred. It is recommended that the **MEX** option (see the **OPT** directive) be used to examine structured control statement expansion for relevant constructs to determine default register usage and applicable reassignment strategies.

See also: **OPT** (**MEX**), **SCSJMP**

A label is not allowed with this directive.

EXAMPLE:

```
        SCSREG  Y0,B            ; reassign SCS source and dest. registers
```

SECTION
Start Section

```
SECTION     <symbol>     [GLOBAL | STATIC | LOCAL]
.
.
<section source statements>
.
.
ENDSEC
```

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 <section name> 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 <section name> 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 <section name> 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 (see the **OPT** directive, this chapter).

Sections may also be split into separate parts. That is, <section name> can be used multiple times with **SECTION** and **ENDSEC** directive pairs. If this occurs, then these separate (but identically named) sections can access each others 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.

Sections may be relocatable or absolute. In the Assembler absolute mode (command line **-A** option) all sections are considered absolute. A full set of locations counters is reserved for each absolute section unless the **GS** option is given (see the **OPT** directive, this chapter). 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.

See also: **MODE**, **ORG**, **GLOBAL**, **LOCAL**, **XDEF**, **XREF**

EXAMPLE:

```
        SECTION   TABLES          ; TABLES will be the section name
```

SET
Set Symbol to a Value

<label> **SET** <expression>

SET <label> <expression>

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** must be absolute and cannot include a symbol that is not yet defined (no forward references are allowed).

See also: **EQU**, **GSET**

EXAMPLE:

```
COUNT   SET     0               ; INITIALIZE COUNT
```

STITLE
Initialize Program Sub-Title

STITLE [<string>]

The **STITLE** directive initializes the program subtitle to the string in the operand field. The subtitle will be printed on the top of all succeeding pages until another **STITLE** directive is encountered. The subtitle is initially blank. The **STITLE** directive will not be printed in the source listing. An **STITLE** directive with no string argument will cause the current subtitle to be blank.

A label is not allowed with this directive.

See also: **TITLE**

EXAMPLE:

```
STITLE      'COLLECT SAMPLES'
```

SYMOBJ
Write Symbol Information to Object File

SYMOBJ <symbol>[,<symbol>,...,<symbol>]

The **SYMOBJ** directive causes information for each <symbol> to be written to the object file. This directive is recognized but currently performs no operation in COFF Assemblers (see Appendix E, Motorola DSP Object File Format (COFF)).

A label is not allowed with this directive.

EXAMPLE:

SYMOBJ XSTART,HIRTN,ERRPROC

TABS
Set Listing Tab Stops

TABS <tabstops>

The **TABS** directive allows resetting the listing file tab stops from the default value of 8.

A label is not allowed with this directive.

See also: **LSTCOL**

EXAMPLE:

```
        TABS    4               ; Set listing file tab stops to 4
```

TITLE
Initialize Program Title

TITLE [<string>]

The **TITLE** directive initializes the program title to the string in the operand field. The program title will be printed on the top of all succeeding pages until another **TITLE** directive is encountered. The title is initially blank. The **TITLE** directive will not be printed in the source listing. A **TITLE** directive with no string argument will cause the current title to be blank.

A label is not allowed with this directive.

See also: **STITLE**

EXAMPLE:

```
        TITLE   'FIR FILTER'
```

UNDEF
Undefine DEFINE Symbol

UNDEF [<symbol>]

The **UNDEF** directive causes the substitution string associated with <symbol> to be released, and <symbol> will no longer represent a valid **DEFINE** substitution. See the **DEFINE** directive for more information.

A label is not allowed with this directive.

See also: **DEFINE**

EXAMPLE:

```
        UNDEF   DEBUG   ; UNDEFINES THE DEBUG SUBSTITUTION STRING
```

WARN
Programmer Generated Warning

WARN [{<str>|<exp>}[,{<str>|<exp>},...,{<str>|<exp>}]]

The **WARN** directive will cause a warning message to be output by the Assembler. The total warning count will be 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 specified optionally to describe the nature of the generated warning.

A label is not allowed with this directive.

See also: **FAIL**, **MSG**

EXAMPLE:

```
        WARN    'parameter too large'
```

XDEF
External Section Symbol Definition

XDEF <symbol>[,<symbol>,...,<symbol>]

The **XDEF** directive is used to specify that the list of symbols is defined within the current section, and that those definitions should be accessible by sections with a corresponding **XREF** directive. This directive is only valid if used within a program section bounded by the **SECTION** and **ENDSEC** directives. The **XDEF** 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 will be generated.

A label is not allowed with this directive.

See also: **SECTION**, **XREF**

EXAMPLE:

```
        SECTION   IO
        XDEF      LOOPA    ; LOOPA will be  accessible by sections with XREF
        .
        .
        .
        ENDSEC
```

XREF
External Section Symbol Reference

XREF <symbol>[,<symbol>,...,<symbol>]

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 will be generated, and all references within the current section to such a symbol will be flagged as undefined. The **XREF** directive must appear before any reference to <symbol> in the section.

A label is not allowed with this directive.

See also: **SECTION**, **XDEF**

EXAMPLE:

```
SECTION   FILTER
XREF      AA,CC,DD      ; XDEFed symbols within section
.
.
.
ENDSEC
```