Freescale DSP
Assembler Reference Manual
Preface

This manual documents the assembler as of version 6.3.28 of the software.

Notation

The notational conventions used in this manual are:

**DIRECTIVE**

All assembler mnemonics and directives are shown in bold upper case to highlight them. However, the assembler will recognize both upper and lower case for mnemonics and directives.

{ }

Contains a list of elements or directives, one of which must be selected. Each choice will be separated by a vertical bar. For example, \{R | L\} indicates that either R or L must be selected.

[ ]

Contains one or more optional elements. If more than one optional element is shown, the required element separators are indicated. All elements outside of the angle brackets (< >) must be specified as they appear. For example, the syntactical element [<number>,] requires the comma to be specified if the optional element <number> is selected.
Preface

The element names are printed in lower case and contained in angle brackets. Some common elements used to describe directives are:

- `<comment>` A statement comment
- `<label>` A statement label
- `<expr>` or `<expression>` An assembler expression
- `<number>` A numeric constant
- `<string>` A string of ASCII characters enclosed in quotes
- `<delimiter>` A delimiter character
- `<option>` An assembler option
- `<sym>` or `<symbol>` An assembler symbol

Supporting Publications


## Table of Contents

### Preface

### Chapter 1 Freescale DSP Assembler

1.1 Introduction ............................................................................................................ 1  
1.2 Assembly Language .............................................................................................. 1  
1.3 Installing the Assembler ....................................................................................... 1  
1.4 Running the Assembler ........................................................................................ 1  
1.5 Assembler Processing ............................................................................................ 8  
1.6 Definition of Terms .............................................................................................. 8  
1.7 Assembler Support for Digital Signal Processing .............................................. 9  

### Chapter 2 Writing Assembly Language Programs

2.1 Input File Format .................................................................................................. 11  
2.2 Symbol Names ..................................................................................................... 11  
2.3 Strings ................................................................................................................ 11  
2.4 Source Statement Format ..................................................................................... 12  
  2.4.1 Label Field .................................................................................................... 13  
  2.4.2 Operation Field ............................................................................................. 14  
  2.4.3 Operand Field ................................................................................................ 14  
  2.4.4 Data Transfer Fields ...................................................................................... 14  
  2.4.5 Comment Field .............................................................................................. 15  
2.5 Assembler Output ................................................................................................. 15
## Table of Contents (Continued)

### Chapter 3  Expressions

3.1 Introduction .......................................................................................... 17
3.2 Absolute and Relative Expressions ......................................................... 17
3.3 Expression Memory Space Attribute ....................................................... 17
3.4 Internal Expression Representation ......................................................... 19
3.5 Constants .............................................................................................. 19
3.5.1 Numeric Constants ............................................................................. 19
3.5.2 String Constants ................................................................................. 20
3.6 Operators .............................................................................................. 20
3.6.1 Unary operators ............................................................................... 20
3.6.2 Arithmetic operators ....................................................................... 21
3.6.3 Shift operators ................................................................................. 21
3.6.4 Relational operators ....................................................................... 22
3.6.5 Bitwise operators ............................................................................. 22
3.6.6 Logical operators ............................................................................. 22
3.7 Operator Precedence ............................................................................. 23
3.8 Functions .............................................................................................. 23
3.8.1 Mathematical Functions .................................................................. 24
3.8.2 Conversion Functions ..................................................................... 25
3.8.3 String Functions .............................................................................. 25
3.8.4 Macro Functions .............................................................................. 25
3.8.5 Assembler Mode Functions ............................................................... 26

### Chapter 4  Software Project Management

4.1 Introduction .......................................................................................... 37
4.2 Sections ............................................................................................... 37
4.3 Sections and Data Hiding ................................................................. 38
4.3.1 Sections and Symbols ................................................................. 38
4.3.2 Sections and Macros ................................................................. 40
4.3.3 Nested and Fragmented Sections ................................................... 40
4.4 Sections and Relocation ................................................................... 41
Table of Contents (Continued)

4.5 Address Assignment ............................................................................................ 42
  4.5.1 The ORG Directive ....................................................................................... 43
  4.5.2 Overlays ....................................................................................................... 45
  4.5.3 Address Assignment Examples .................................................................... 46
  4.5.4 Circular Buffers ........................................................................................... 47
4.6 Example 1: Multi-Programmer Environment ......................................................... 48
  4.6.1 Absolute Mode Implementation .................................................................. 49
  4.6.2 Relative Mode Implementation ................................................................ 50
4.7 Example 2: Overlays ........................................................................................... 51
  4.7.1 Absolute Mode Implementation .................................................................. 52
  4.7.2 Relative Mode Implementation ................................................................ 53
4.8 Example 3: Bootstrap Overlay ............................................................................. 54
  4.8.1 Absolute Mode Implementation .................................................................. 55
  4.8.2 Relative Mode Implementation ................................................................ 56

**Chapter 5 Macro Operations and Conditional Assembly**

  5.1 Macro Operations .............................................................................................. 57
  5.2 Macro Libraries .................................................................................................. 58
  5.3 Macro Definition ................................................................................................ 58
  5.4 Macro Calls ........................................................................................................ 60
  5.5 Dummy Argument Operators ............................................................................ 61
    5.5.1 Dummy Argument Concatenation Operator - \ ........................................ 61
    5.5.2 Return Value Operator - ? ........................................................................ 62
    5.5.3 Return Hex Value operator - % .................................................................. 63
    5.5.4 Dummy Argument String Operator - " ...................................................... 63
    5.5.5 Macro Local Label Override Operator - ^ ................................................ 64
  5.6 DUP, DUPA, DUPC, DUPF Directives ................................................................ 65
  5.7 Conditional Assembly ....................................................................................... 65

**Chapter 6 Assembler Significant Characters and Directives**

  6.1 Introduction ....................................................................................................... 67
Table of Contents (Continued)

6.2 Assembler Significant Characters ................................................................. 67
6.3 Assembler Directives ....................................................................................... 68
   6.3.1 Assembly Control ................................................................................... 68
   6.3.2 Symbol Definition ................................................................................. 69
   6.3.3 Data Definition/Storage Allocation ...................................................... 69
   6.3.4 Listing Control and Options ................................................................. 69
   6.3.5 Object File Control ............................................................................... 70
   6.3.6 Macros and Conditional Assembly ..................................................... 70
   6.3.7 Structured Programming ..................................................................... 70

Chapter 7 Structured Control Statements

7.1 Introduction ................................................................................................... 167
7.2 Structured Control Directives ....................................................................... 167
7.3 Syntax ........................................................................................................... 168
   7.3.1 .BREAK Statement ............................................................................ 168
   7.3.2 .CONTINUE Statement ..................................................................... 169
   7.3.3 .FOR Statement .............................................................................. 170
   7.3.4 .IF Statement ................................................................................. 171
   7.3.5 .LOOP Statement ............................................................................ 172
   7.3.6 .REPEAT Statement ....................................................................... 173
   7.3.7 .WHILE Statement ........................................................................... 173
7.4 Simple and Compound Expressions ............................................................. 174
   7.4.1 Simple Expressions .......................................................................... 174
   7.4.1.1 Condition Code Expressions ...................................................... 174
   7.4.1.2 Operand Comparison Expressions ............................................ 175
   7.4.2 Compound Expressions .................................................................... 176
7.5 Statement Formatting ................................................................................... 176
   7.5.1 Expression Formatting ....................................................................... 176
   7.5.2 .FOR/.LOOP Formatting ................................................................. 177
   7.5.3 Assembly Listing Format ................................................................... 177
7.6 Effects on the Programmer’s Environment ................................................... 177
# Table of Contents (Continued)

## Appendix A  ASCII Character Codes

## Appendix B  Directive Summary

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.1 Assembly Control</td>
<td>181</td>
</tr>
<tr>
<td>B.2 Symbol Definition</td>
<td>182</td>
</tr>
<tr>
<td>B.3 Data Definition/Storage Allocation</td>
<td>182</td>
</tr>
<tr>
<td>B.4 Listing Control and Options</td>
<td>182</td>
</tr>
<tr>
<td>B.5 Object File Control</td>
<td>183</td>
</tr>
<tr>
<td>B.6 Macros and Conditional Assembly</td>
<td>183</td>
</tr>
<tr>
<td>B.7 Structured Programming</td>
<td>183</td>
</tr>
</tbody>
</table>

## Appendix C  Assembler Messages

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.1 Introduction</td>
<td>185</td>
</tr>
<tr>
<td>C.2 Command Line Errors</td>
<td>186</td>
</tr>
<tr>
<td>C.3 Warnings</td>
<td>188</td>
</tr>
<tr>
<td>C.4 Errors</td>
<td>196</td>
</tr>
<tr>
<td>C.5 Fatal Errors</td>
<td>226</td>
</tr>
</tbody>
</table>

## Appendix D  Freescale DSP Object File Format (COFF)

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>D.1 Introduction</td>
<td>231</td>
</tr>
<tr>
<td>D.2 Object File Structure</td>
<td>231</td>
</tr>
<tr>
<td>D.3 Object File Components</td>
<td>233</td>
</tr>
<tr>
<td>D.3.1 File Header</td>
<td>233</td>
</tr>
<tr>
<td>D.3.2 Optional Header</td>
<td>234</td>
</tr>
<tr>
<td>D.3.3 Sections</td>
<td>236</td>
</tr>
<tr>
<td>D.3.3.1 Section Headers</td>
<td>237</td>
</tr>
<tr>
<td>D.3.3.2 Relocation Information</td>
<td>239</td>
</tr>
<tr>
<td>D.3.3.3 Line Numbers</td>
<td>240</td>
</tr>
<tr>
<td>D.3.4 Symbol Table</td>
<td>241</td>
</tr>
<tr>
<td>D.3.4.1 Symbol Name</td>
<td>243</td>
</tr>
</tbody>
</table>
Table of Contents (Continued)

D.3.4.2 Symbol Value ................................................................. 243
D.3.4.3 Section Number ............................................................... 244
D.3.4.4 Symbol Type ................................................................. 244
D.3.4.5 Symbol Storage Class ....................................................... 246
D.3.4.6 Auxiliary Entries ............................................................. 250
D.3.4.6.1 Filenames ............................................................... 251
D.3.4.6.2 Sections ................................................................. 251
D.3.4.6.3 Tag Names ............................................................... 253
D.3.4.6.4 End of Structures ....................................................... 253
D.3.4.6.5 Functions ................................................................. 254
D.3.4.6.6 Arrays ............................................................... 255
D.3.4.6.7 End of Blocks and Functions ....................................... 255
D.3.4.6.8 Beginning of Blocks and Functions .............................. 256
D.3.4.6.9 Structure, Union, and Enumeration Names ................. 256
D.3.4.7 Object File Comments ..................................................... 257
D.3.5 String Table ................................................................. 257
D.4 Differences in DSP Object Format and Standard COFF ............... 257
D.4.1 Multiple Memory Spaces .................................................. 258
D.4.2 Object File Transportability ................................................ 259
D.4.3 Structure Size Fields ....................................................... 260
D.4.4 Relocation Information ...................................................... 260
D.4.5 Block Data Sections ......................................................... 261
D.4.6 Other Extensions ............................................................. 261
D.5 Object File Data Expression Format ......................................... 261
D.5.1 Data Expression Generation ............................................... 262
D.5.2 Data Expression Interpretation ............................................ 262
D.5.2.1 User Expression - { ... } ............................................... 263
D.5.2.2 Relocatable Expression - [ ... ] ....................................... 263
D.5.2.3 Memory Space Operator - @ ........................................ 263
D.5.2.4 Bit Size Operator - # .................................................... 263
D.5.2.5 Memory Attribute Operator - : ...................................... 264
D.5.2.6 Line Number Operator - ! ............................................. 264
### Table of Contents (Continued)

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>D.5.2.7</td>
<td>BFxxx Instruction Mask Function - @FBF()</td>
<td>264</td>
</tr>
<tr>
<td>D.5.2.8</td>
<td>Local Relocatable Reference Function - @LRF()</td>
<td>264</td>
</tr>
<tr>
<td>D.5.2.9</td>
<td>Alternate Encoding Function - @ENC()</td>
<td>265</td>
</tr>
</tbody>
</table>

### Appendix E  Instruction Set Information

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>E.1</td>
<td>DSP56300 Information</td>
<td>267</td>
</tr>
<tr>
<td>E.1.1</td>
<td>Instruction Set Summary</td>
<td>267</td>
</tr>
<tr>
<td>E.1.1.1</td>
<td>Arithmetic Instructions</td>
<td>267</td>
</tr>
<tr>
<td>E.1.1.2</td>
<td>Logical Instructions</td>
<td>268</td>
</tr>
<tr>
<td>E.1.1.3</td>
<td>Bit Manipulation Instructions</td>
<td>268</td>
</tr>
<tr>
<td>E.1.1.4</td>
<td>Loop Instructions</td>
<td>269</td>
</tr>
<tr>
<td>E.1.1.5</td>
<td>Move Instructions</td>
<td>269</td>
</tr>
<tr>
<td>E.1.1.6</td>
<td>Program Control Instructions</td>
<td>269</td>
</tr>
<tr>
<td>E.1.2</td>
<td>Register Names and Usage</td>
<td>270</td>
</tr>
<tr>
<td>E.1.3</td>
<td>Condition Code Mnemonics</td>
<td>270</td>
</tr>
</tbody>
</table>

### Index
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-1</td>
<td>COFF File Basic Structure</td>
<td>232</td>
</tr>
<tr>
<td>A-2</td>
<td>File Header Format</td>
<td>233</td>
</tr>
<tr>
<td>A-3</td>
<td>File Header Flags</td>
<td>234</td>
</tr>
<tr>
<td>A-4</td>
<td>Freescale DSP Optional Link Header Format</td>
<td>235</td>
</tr>
<tr>
<td>A-5</td>
<td>Freescale DSP Optional Runtime Header Format</td>
<td>236</td>
</tr>
<tr>
<td>A-6</td>
<td>Section Header Format</td>
<td>237</td>
</tr>
<tr>
<td>A-7</td>
<td>Section Header Flags</td>
<td>239</td>
</tr>
<tr>
<td>A-8</td>
<td>Relocation Entry Format</td>
<td>240</td>
</tr>
<tr>
<td>A-9</td>
<td>Line Number Entry Format</td>
<td>240</td>
</tr>
<tr>
<td>A-10</td>
<td>Line Number Grouping</td>
<td>241</td>
</tr>
<tr>
<td>A-11</td>
<td>COFF Symbol Table Ordering</td>
<td>242</td>
</tr>
<tr>
<td>A-12</td>
<td>Symbol Table Entry Format</td>
<td>243</td>
</tr>
<tr>
<td>A-13</td>
<td>Fundamental Types</td>
<td>245</td>
</tr>
<tr>
<td>A-14</td>
<td>Derived Types</td>
<td>246</td>
</tr>
<tr>
<td>A-15</td>
<td>Storage Classes</td>
<td>247</td>
</tr>
<tr>
<td>A-16</td>
<td>Storage Class and Value</td>
<td>249</td>
</tr>
<tr>
<td>A-17</td>
<td>Filename Symbol Auxiliary Entry</td>
<td>251</td>
</tr>
<tr>
<td>A-18</td>
<td>Section Symbol Auxiliary Entry</td>
<td>251</td>
</tr>
<tr>
<td>A-19</td>
<td>Relocatable Section Auxiliary Entry</td>
<td>252</td>
</tr>
<tr>
<td>A-20</td>
<td>Relocatable Buffer/Overlay Auxiliary Entry</td>
<td>252</td>
</tr>
<tr>
<td>A-21</td>
<td>Tag Name Symbol Auxiliary Entry</td>
<td>253</td>
</tr>
<tr>
<td>A-22</td>
<td>End of Structure Auxiliary Entry</td>
<td>254</td>
</tr>
<tr>
<td>A-23</td>
<td>Function Symbol Auxiliary Entry</td>
<td>254</td>
</tr>
<tr>
<td>A-24</td>
<td>Array Symbol Auxiliary Entry</td>
<td>255</td>
</tr>
<tr>
<td>A-25</td>
<td>End of Block or Function Auxiliary Entry</td>
<td>255</td>
</tr>
<tr>
<td>A-26</td>
<td>Beginning of Block or Function Auxiliary Entry</td>
<td>256</td>
</tr>
<tr>
<td>A-27</td>
<td>Structure, Union, or Enumeration Name Auxiliary Entry</td>
<td>257</td>
</tr>
<tr>
<td>A-28</td>
<td>CORE_ADDR Format</td>
<td>258</td>
</tr>
<tr>
<td>A-29</td>
<td>Memory Mapping Enumerations</td>
<td>259</td>
</tr>
<tr>
<td>A-30</td>
<td>Freescale DSP COFF Byte Ordering</td>
<td>260</td>
</tr>
</tbody>
</table>
Chapter 1 Freescale DSP Assembler

1.1 Introduction

The Freescale DSP Assemblers are programs that process assembly language source statements written for Freescale’s family of digital signal processors. The assembler translates these source statements into object programs compatible with other Freescale DSP software and hardware products.

1.2 Assembly Language

The assembly language provides mnemonic operation codes for all machine instructions in the digital signal processor instruction set. In addition, the assembly language contains mnemonic directives which specify auxiliary actions to be performed by the assembler. These directives are not always translated into machine language. The assembly language enables the programmer to define and use macro instructions which replace a single statement with a predefined sequence of statements found in the macro definition. Conditional assembly also is supported.

1.3 Installing the Assembler

The assembler is distributed on various media and in different formats depending on the host environment.

1.4 Running the Assembler

The general format of the command line to invoke the assembler is:

    ASM56300 [options] <filenames>

where:

ASM56300

The name of the Freescale DSP assembler program appropriate for the target processor. For the Freescale DSP56300 processor, the name of the assembler executable is ASM56300.
Any of the following command line options. These can be in any order, but must precede the list of source filenames. Some options can be given more than once; the individual descriptions indicate which options may be specified multiple times. Option letters can be in either upper or lower case.

Command options that are used regularly may be placed in the environment variable **DSPASMOPT**. If the variable is found in the environment the assembler adds the associated text to the existing command line prior to processing any options. See your host documentation for instructions on how to define environment variables.

Option arguments may immediately follow the option letter or may be separated from the option letter by blanks or tabs. However, an ambiguity arises if an option takes an optional argument. Consider the following command line:

```
ASM56300  -B MAIN IO
```

In this example it is not clear whether the file MAIN is a source file or is meant to be an argument to the `-B` option. If the ambiguity is not resolved the assembler will assume that MAIN is a source file and attempt to open it for reading. This may not be what the programmer intended.

There are several ways to avoid this ambiguity. If MAIN is supposed to be an argument to the `-B` option it can be placed immediately after the option letter:

```
ASM56300  -BMAIN IO
```

If there are other options on the command line besides those that take optional arguments the other options can be placed between the ambiguous option and the list of source file names:

```
ASM56300  -B MAIN -V IO
```

An alternative is to use two successive hyphens to indicate the end of the option list:

```
ASM56300  -B -- MAIN IO
```

In this latter case the assembler interprets MAIN as a source file name and uses the default naming conventions for the `-B` option.

**-A**

Indicates that the assembler should run in absolute mode, generating an absolute object file when the `-B` command line option is given. By default the assembler produces a relocatable object file that is subsequently processed.
by the Freescale DSP linker. See Chapter 4, for more information on assembler modes.

-B[<objfil>]

This option specifies that an object file is to be created for assembler output. <objfil> can be any legal operating system filename, including an optional pathname. A hyphen also may be used as an argument to indicate that the object file should be sent to the standard output.

The type of object file produced depends on the assembler operation mode. If the -A option is supplied on the command line, the assembler operates in absolute mode and generates an absolute object (.CLD) file. If there is no -A option on the command line, the assembler operates in relative mode and creates a relocatable object (.CLN) file.

If a pathname is not specified, the file will be created in the current directory. If no filename is specified, the assembler will use the basename (filename without extension) of the first filename encountered in the source input file list and append the appropriate file type (.CLN or .CLD) to the basename. If the -B option is not specified, then the assembler will not generate an object file. The -B option should be specified only once. If the file named in the -B option already exists, it will be overwritten.

Example: ASM56300 -Bfilter main.asm fft.asm fio.asm

In this example, the files MAIN.ASM, FFT.ASM, and FIO.ASM are assembled together to produce the relocatable object file FILTER.CLN.

-D<symb> <string>

This is equivalent to a source statement of the form:

DEFINE <symbol> <string>

<string> must be preceded by a blank and should be enclosed in single quotes if it contains any embedded blanks. Note that if single quotes are used they must be passed to the assembler intact, e.g. some host command interpreters will strip single quotes from around arguments. The -D<symb> <string> sequence can be repeated as often as desired. See the DEFINE directive (Chapter 6) for more information.

Example: ASM56300 -D POINTS 16 prog.asm

All occurrences of the symbol POINTS in the program PROG.ASCII will be replaced by the string ‘16’.
Chapter 1 Freescale DSP Assembler - Running the Assembler

-**EA**<errfil>
-**EW**<errfil>

These options allow the standard error output file to be reassigned on hosts that do not support error output redirection from the command line. <errfil> must be present as an argument, but can be any legal operating system filename, including an optional pathname.

The **-EA** option causes the standard error stream to be written to <errfil>; if <errfil> exists, the output stream is appended to the end of the file. The **-EW** option also writes the standard error stream to <errfil>; if <errfil> exists it is rewound (truncated to zero), and the output stream is written from the beginning of the file.

Example: **ASM56300  -EW**errors prog.asm

Redirect the standard error output to the file ERRORS. If the file already exists, it will be overwritten.

-**F**<argfil>

Indicates that the assembler should read command line input from <argfil>. <argfil> can be any legal operating system filename, including an optional pathname. <argfil> is a text file containing further options, arguments, and filenames to be passed to the assembler. The arguments in the file need be separated only by some form of white space (blank, tab, newline). A semicolon (;) on a line following white space makes the rest of the line a comment.

The **-F** option was introduced to circumvent the problem of limited line lengths in some host system command interpreters. It may be used as often as desired, including within the argument file itself. Command options may also be supplied using the **DSPASMOPT** environment variable. See the discussion of **DSPASMOPT** under [options] at the beginning of this section.

Example: **ASM56300  -F**opts.cmd

Invoke the assembler and take command line options and source filenames from the command file OPTS.CMD.

-**G**

Send source file line number information to the object file. This option is valid only in conjunction with the **-B** command line option. The generated line
number information can be used by debuggers to provide source-level debugging.

Example: **ASM56300 -B -G myprog.asm**

Assemble the file MYPROG.ASM and send source file line number information to the resulting object file MYPROG.CLN.

`-I<pathname>`

When the assembler encounters **INCLUDE** files, the current directory (or the directory specified in the **INCLUDE** directive) is first searched for the file. If it is not found and the `-I` option is specified, the assembler prefixes the filename (and optional pathname) specified in the **INCLUDE** directive with `<pathname>` and searches the newly formed directory pathname for the file.

The pathname must be a legal operating system pathname. The `-I` option may be repeated as many times as desired. The directories will be searched in the order specified on the command line.

Example: **ASM56300 -Iproject\ testprog**

This example uses IBM PC pathname conventions, and would cause the assembler to prefix any **INCLUDE** files not found in the current directory with the \project\ pathname.

`-L<lstfil>`

This option specifies that a listing file is to be created for assembler output. `<lstfil>` can be any legal operating system filename, including an optional pathname. A hyphen also may be used as an argument to indicate that the listing file should be sent to the standard output, although the listing file is routed to standard output by default.

If a pathname is not specified, the file will be created in the current directory. If no filename is specified, the assembler will use the basename (filename without extension) of the first filename encountered in the source input file list and append .LST to the basename. If the `-L` option is not specified, then the assembler will route listing output to the standard output (usually the console or terminal screen) by default. The `-L` option should be specified only once. **If the file named in the -L option already exists, it will be overwritten.**

Example: **ASM56300 -L filter.asm gauss.asm**

In this example, the files FILTER.ASM and GAUSS.ASM are assembled together to produce a listing file. Because no filename was given with the `-L` option, the output file will be named using the

`-L`
basename of the first source file, in this case FILTER. The listing file will be called FILTER.LST.

-M<pathname>

This is equivalent to a source statement of the form:

MACLIB <pathname>

The pathname must be a legal operating system pathname. The -M option may be repeated as many times as desired. The directories will be searched in the order specified on the command line. See the MACLIB directive (Chapter 6) for more information.

Example: ASM56300 -M fftlib/ trans.asm

This example uses UNIX pathname conventions, and would cause the assembler to look in the fftlib subdirectory of the current directory for a file with the name of the currently invoked macro found in the source file.

-O<opt>[,<opt>,...,<opt>]

This is equivalent to a source statement of the form:

OPT <opt>[,<opt>,...,<opt>]

<opt> can be any of the options that are available with the OPT directive (see Chapter 6). If multiple options are specified, they must be separated by commas. The -O<opt> sequence can be repeated for as many options as desired.

Example: ASM56300 -OS,CRE myprog.asm

This will activate the symbol table and cross reference listing options.

-Q

On some hosts the assembler displays a banner on the console when invoked. This option inhibits the banner display. It has no effect on hosts where the signon banner is not displayed by default.

Example: ASM56300 -Q myprog.asm

Assemble the file MYPROG.ASM but do not display the signon banner on the console.
This option causes the assembler to report assembly progress (beginning of passes, opening and closing of input files) to the standard error output stream. This is useful to insure that assembly is proceeding normally.

Example: ASM56300 -V myprog.asm

Assemble the file MYPROG.ASM and send progress lines to the standard error output.

-Z

This option causes the assembler to strip symbol information from the absolute load file. Normally symbol information is retained in the object file for symbolic reference purposes. Note that this option is valid only when the assembler is in absolute mode via the -A command line option and when an object file is created (-B option).

Example: ASM56300 -A -B -Z myprog.asm

Assemble the file MYPROG.ASM in absolute mode and strip symbol information from the load file created as output.

<filenames>

A list of operating system compatible filenames (including optional pathnames). If no extension is supplied for a given file, the assembler first will attempt to open the file using the filename as supplied. If that is not successful the assembler appends .ASM to the filename and attempts to open the file again. If no pathname is specified for a given file, the assembler will look for that file in the current directory. The list of files will be processed sequentially in the order given and all files will be used to generate the object file and listing.

The assembler will redirect the output listing to the standard output if the output listing is not suppressed with the IL option, or if it is not redirected via the -L command line option described above. The standard output generally goes to the console or terminal screen by default, but can be diverted to a file or to a printer by using the I/O redirection facilities of the host operating system, if available. Error messages will always appear on the standard output, regardless of any option settings. Note that some options (-B, -L) allow a hyphen as an optional argument which indicates that the corresponding output should be sent to the standard output stream. Unpredictable results may occur if, for example, the object file is explicitly routed to standard output while the listing file is allowed to default to the same output stream.
1.5 Assembler Processing

The Freescale DSP assembler is a two-pass assembler. During the first pass the source program is read to build the symbol and macro tables. During the second pass the object file is generated (assembled) with reference to the tables created during pass one. It is also during the second pass that the source program listing is produced.

Each source statement is processed completely before the next source statement is read. As each line is read in, any translations specified by the `DEFINE` directive are applied. Each statement is then processed, and the assembler examines the label, operation code, operand, and data transfer fields. The macro definition table is scanned for a match with the operation code. If there is no match, the operation code and directive tables are scanned for a match with a known opcode.

Any errors detected by the assembler are displayed before the actual line containing the error is printed. Errors and warnings are accumulated, and a total number of errors and warnings is printed at the end of the source listing. If no source listing is produced, error messages are still displayed to indicate that the assembly process did not proceed normally. The number of errors is returned as an exit status when the assembler returns control to the host operating system.

1.6 Definition of Terms

Since the Freescale DSP architectures are different from normal microprocessors, the programmer may not be familiar with some of the terms used in this document. The following discussion serves to clarify some of the concepts discussed later in this manual.

The Freescale DSP architecture can have as many as five separate memory spaces referred to as the \( X, Y, L, P \) (Program). \( L \) memory space is a concatenation of \( X \) and \( Y \) data memory and is considered by the assembler as a superset of the \( X \) and \( Y \) memory spaces. The assembler will generate object code for each memory space, but object code can only be generated for one memory space at a time.

The memory space and address location into which the object code generated by the assembler will be loaded are referred to as the **load memory space** and **load address**, respectively. Because the DSP architecture allows data transfers between memory spaces, sometimes object code is loaded into an address of one memory space but will later be transferred to a different memory space and address before the program is run. One example of this might be a program located in an external EPROM that will be transferred into external program RAM before it is run. The transfer of code/data from one memory space/address to a different memory space/address is called an **overlay**.

When the object code for a part of the program is generated that later will be used as an overlay, the load memory space and load address do not correspond to the memory space and address where the program will be run. The memory space and address location where the code/data will be located when the program is run are referred to as the **runtime memory space** and **runtime address**, respectively. If the assembler only used the load address to assign values to labels, then the program would not contain the cor-
rect label references when it was transferred to the runtime memory space and the runtime address.

During the assembly process, the assembler uses location counters to record the addresses associated with the object code. In order to facilitate the generation of object code for overlays, the assembler maintains two different location counters, the load location counter, which determines the address into which the object code will be loaded and the runtime location counter, which determines the address assigned to labels. In addition, the assembler keeps track of the load memory space, which is the memory space into which the object code will be loaded, and the runtime memory space, which is the memory space to which an overlay will be transferred and the memory space attribute that will be assigned to labels. See Chapter 4, for a practical discussion of the use of memory spaces and location counters.

The Freescale digital signal processors are capable of performing operations on modulo and reverse-carry buffers, two data structures useful in digital signal processing applications. The DSP assembler provides directives for establishing buffer base addresses, allocating buffer space, and initializing buffer contents. For a buffer to be located properly in memory the lower bits of the starting address which encompass one less than the buffer size must be zero. For example, the lowest address greater than zero at which a buffer of size 32 may be located is 32 (20 hexadecimal). More generally, the buffer base address must be a multiple of $2^k$, where $2^k$ is greater than or equal to the size of the buffer. Buffers can be allocated manually or by using the assembler buffer directives (see Chapter 6).

The assembler operates in either absolute or relative mode, depending on the presence of the command line -A option. In relative mode the assembler creates relocatable object files. These files can be combined and relocated using the Freescale DSP linker. In absolute mode the assembler generates absolute object files. Absolute files cannot be relocated but can be loaded directly for execution. By default the assembler runs in relative mode.

1.7 Assembler Support for Digital Signal Processing

As mentioned previously, the assembler offers facilities commonly found in other macro assemblers, such as nested macro capabilities, include files, and conditional assembly. The assembler must also provide extensions in support of the unconventional architecture of the Freescale digital signal processors, as well as aids for programming DSP-specific applications. Some of these features are discussed briefly below; see the appropriate chapters later in this manual for more information.

The assembler supports the use of arbitrary algebraic expressions as arguments to various directives and as immediate operands in certain instructions. Terms of these expressions may consist of the assembler’s own built-in functions, which perform data conversion, comparison, and computational operations. In the digital signal processing domain transcendental functions for computing sine, cosine, and natural logarithm are useful for initializing data values in memory, such as sine/cosine tables for FFT algorithms. Also, there are functions for easily converting values expressed in decimal float-
ing point to their binary or fractional equivalents. This conversion is done automatically for immediate instruction operands and arguments to the DC directive (see Chapter 6). See Chapter 3 for more information on assembler expressions, operators, and built-in functions.

The register set of the Freescale digital signal processors allows for efficient use of modulo and reverse-carry buffers for FFT applications. The assembler supports this architecture by providing several special-purpose directives for allocating circular buffers. The BADDR, BUFFER, DSM, and DSR directives automatically advance the program counter to the next appropriate base address given the buffer size, and perform various boundary and magnitude checks to insure that the buffer is valid. The BSM and BSR provide for automatic alignment and block initialization of DSP buffers. Since a buffer allocated in this fashion can cause alignment gaps in memory, the MU option (see the OPT directive, Chapter 6) may be used to generate a full memory utilization report. See Chapter 6 for more information on assembler directives and options.
Chapter 2 Writing Assembly Language Programs

2.1 Input File Format

Programs written in assembly language consist of a sequence of source statements. Any source statement can be extended to one or more lines by including the line continuation character (\) as the last character on the line to be continued. A source statement (first line and any continuation lines) can be a maximum of 255 characters long. Upper and lower case letters are considered equivalent for assembler mnemonics and directives, but are considered distinct for labels, symbols, directive arguments, and literal strings.

If the source file contains horizontal tabs (ASCII $09), the assembler will expand these to the next fixed tab stop located at eight character intervals (column 1, 9, 17...), unless reset using the TAB directive (see Chapter 6). This is only significant if tab characters are embedded within literal strings.

2.2 Symbol Names

Symbol names can be from one to 512 characters long. The first character of a symbol must be alphabetic (upper or lower case); any remaining characters can be either alphanumeric (A-Z, a-z, 0-9) or the underscore character (_). Upper and lower case letters in symbols are considered distinct unless the IC option is in effect (see the OPT directive, Chapter 6).

Valid: loop_1
ENTRY
a_B_c

Invalid: 1_loop
loop.e

Certain identifiers are reserved by the assembler and cannot be used. These identifiers are the upper or lower case name of any Freescale DSP processor register. See Appendix E for a list of the register names of the appropriate target processor.

2.3 Strings

One or more ASCII characters enclosed by single quotes (') constitute a literal ASCII string. In order to specify an apostrophe within a literal string, two consecutive apostro-
phes must appear where the single apostrophe is intended. Strings are used as operands for some assembler directives and also can be used to a limited extent in expressions.

A string may also be enclosed in double quotes (""") in which case any DEFINE directive symbols contained in the string would be expanded. The double quote should be used with care inside macros since it is used as a dummy argument string operator (see Chapter 5). In that case the macro concatenation operator can be used to escape the double-quoted string if desired.

Two strings separated by the string concatenation operator (++) will be recognized by the assembler as equivalent to the concatenation of the two strings. For example, these two strings are equivalent:

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

The assembler has a substring extraction capability using the square brackets ([]). Here is an example:

['DSP56300',3,5] = '56300'

Substrings may be used wherever strings are valid and can be nested. There are also functions for determining the length of a string and the position of one string within another. See Chapter 3 for more information on string functions.

### 2.4 Source Statement Format

Each source statement may include up to six fields separated by one or more spaces or tabs: a label field, an operation field, an operand field, up to two data transfer fields, and a comment field. Only fields preceding the comment field are considered significant to the assembler; the comment field is ignored. For example, the following source statement shows all six possible fields for the DSP56300:

<table>
<thead>
<tr>
<th>Label</th>
<th>Opcode</th>
<th>X field</th>
<th>Y field</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENT</td>
<td>FMPY</td>
<td>D8,D6,D2</td>
<td>X:(R0),D4.S</td>
<td>D2,S.Y:(R5)+:</td>
</tr>
</tbody>
</table>

In general, the contents of each field other than the comment field cannot contain embedded whitespace characters, since these characters are used as field delimiters. Two ex-
ceptions are blanks and tabs in quoted strings and the syntax of structured control statements (see Chapter 7).

2.4.1 Label Field

The label field occurs as the first field of a source statement, and can take one of the following forms:

1. A space or tab as the first character on a line ordinarily indicates that the label field is empty, and that the line has no label.

2. An alphabetic character as the first character indicates that the line contains a symbol called a label.

3. An underscore (_ _) as the first character indicates that the label is a local label.

Labels may be indented if the label symbol is immediately followed by a colon (:). If the first non-blank field on a line complies with either forms 2 or 3 above and the field ends with a colon, the assembler regards this as the label field, even if it does not start with the first character on the line. However, all characters preceding the label on the source line must be whitespace characters (spaces or tab characters). There should be no intervening blanks or tabs between the end of the label symbol and the appended colon character.

Local labels are any normal symbol name preceded (with no intervening blanks) by an underscore (_ _). Except for the special case of macros (described below), local labels have a limited scope bounded by any two non-local labels. The local label can be referred to or defined only in source statements that are between two source lines containing non-local labels. Local labels are useful in defining program locations where a unique label name is required but is not considered useful in documenting the source file (for example, the terminating address of a DO loop). Note that the maximum length of a local label includes the leading underscore (_ _) character.

Use of local labels in macros represents a special case. All local labels within a macro are considered distinct for the currently active level of macro expansion (unless the macro local label override operator is used; see Chapter 5). These local labels are valid for the entire macro expansion and are not considered bounded by non-local labels. Therefore, all local labels within a macro must be unique. This mechanism allows the programmer to freely use local labels within a macro definition without regard to the number of times that the macro is expanded. Non-local labels within a macro expansion are considered to be normal labels and therefore cannot occur more than once unless used with the SET directive (see Chapter 6).

A label may occur only once in the label field of an individual source file unless it is used as a local label, a label local to a section, or is used with the SET directive. If a non-local label does occur more than once in a label field, each reference to that label after the first will be flagged as an error.
A line consisting of a label only is a valid line and has the effect of assigning the value of the location counter to the label. With the exception of some directives, a label is assigned the value of the location counter of the first word of the instruction or data being assembled.

2.4.2 Operation Field

The operation field appears after the label field, and must be preceded by at least one space or tab. Entries in the operation field may be one of three types:

- **Opcode** - Mnemonics that correspond directly to DSP machine instructions.
- **Directive** - Special operation codes known to the assembler which control the assembly process.
- **Macro call** - Invocation of a previously defined macro which is to be inserted in place of the macro call.

The assembler first searches for operation codes in an internal macro definition table. If no match is found, the table of machine operation codes and assembler directives is searched. If neither of the tables holds the specified operation code, an error message is generated (this sequence can be altered with the `MACLIB` directive). Macro names can therefore replace standard machine operation codes and assembler directives, although a warning will be issued if such a replacement occurs. The warning can be avoided by use of the `RDIRECT` directive. See Chapter 6 for more information on the `MACLIB` and `RDIRECT` directives.

2.4.3 Operand Field

The interpretation of the operand field is dependent on the contents of the operation field. The operand field, if present, must follow the operation field, and must be preceded by at least one space or tab. The operand field may contain a symbol, an expression, or a combination of symbols and expressions separated by commas. There should be no intervening whitespace characters separating operand elements.

The operand field of machine instructions is used to specify the addressing mode of the instruction, as well as the operand of the instruction. The format of the operand field for a particular instruction is given in Appendix A of the User Manual for the DSP in question (e.g., *DSP56300 Family Manual*). The operand fields of assembler directives are described in Chapter 6. The operand fields of macros (Chapter 5) depend on the definition of the macro.

2.4.4 Data Transfer Fields

Most opcodes can specify one or more data transfers to occur during the execution of the instruction. These data transfers are indicated by two addressing mode operands separated by a comma, with no embedded blanks. If two data transfers are specified, they
must be separated by one or more blanks or tabs. See the appropriate DSP User’s Man-
ual for a complete discussion of addressing modes that are applicable to data transfer
specifications.

2.4.5 Comment Field

Comments are not considered significant to the assembler, but can be included in the
source file for documentation purposes. A comment field is composed of any characters
(not part of a literal string) that are preceded by a semicolon (;). 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, un-
less the NOPP option is used (see the OPT directive, Chapter 6). Comments preceded by
two consecutive semicolons (;;) will not be reproduced on the assembler listing and will
not be saved as part of a macro definition.

2.5 Assembler Output

The assembler output consists of an optional listing of the source program and an optional
object file. Appendix D contains the description of the object file format.

The assembly source program listing contains the original source statements, formatted
for easier reading, as well as additional information which is generated by the assembler.
Most lines in the listing correspond directly to a source statement. Lines which do not cor-
respond directly to source statements include page headings, error messages, and ex-
pansions of macro calls or directives such as DC (Define Constant; see Chapter 6).

The assembly listing optionally may contain a symbol table or a cross-reference table of
all non-local symbols appearing in the program. These are always printed after the end of
source input or the END directive (whichever occurs first) if either the symbol table or
cross-reference table options are in effect (see the OPT directive, Chapter 6). The symbol
table contains the name of each symbol, along with its defined value. The cross-reference
table additionally contains the assembler-maintained source line number of every refer-
ence to every non-local symbol (local symbols may be included in the cross-reference list-
ing by using the LOC option; see the OPT directive, Chapter 6).

If the MU option is enabled (see the OPT directive, Chapter 6), the assembler generates
a report of load and runtime memory utilization. The report shows beginning and ending
addresses of allocated memory areas, along with their lengths and associated symbol
names, if applicable. A separate report is generated for each memory space where data
has been reserved for use by the program.

The assembler object file is a binary COFF (Common Object File Format) file, with exten-
sions and adaptations to support symbolic debugging and to make DSP object files trans-
portable among host platforms. COFF is a formal definition for the structure of machine
code files. It is derived from AT&T Unix System V and represents a quasi-de facto stan-
dard for object file formats. Refer to Appendix D for more information on Freescale DSP
COFF structure and layout.
Chapter 3 Expressions

3.1 Introduction

An expression represents a value which is used as an operand in an assembler instruction or directive. An expression is a combination of symbols, constants, operators, and parentheses. Expressions may contain user-defined labels and their associated integer or floating point values, and/or any combination of integers, floating point numbers, or ASCII literal strings. In general, white space (a blank or tab) is not allowed between the terms and operators of an assembler expression. Expressions otherwise follow the conventional rules of algebra and boolean arithmetic.

3.2 Absolute and Relative Expressions

An expression may be either relative or absolute. An absolute expression is one which consists only of absolute terms, or is the result of two relative terms with opposing signs. A relative expression consists of a relative term by itself or only in combination with absolute terms.

When the assembler is operating in relative mode all address expressions must adhere to the above definitions for absolute or relative expressions. This is because only these types of expressions will retain a meaningful value after program relocation. For example, when relative terms are paired with opposing signs, the result is the difference between the two relative terms, which is an absolute value. However, if two positive relative terms are added together the result is unpredictable based on the computed values of the terms at relocation time.

3.3 Expression Memory Space Attribute

A symbol is associated with either an integer or a floating point value which is used in place of the symbol during the expression evaluation. Each symbol also carries a memory space attribute of either X, Y, L, Program, or None. SET labels, constants, and floating point expressions always have a memory space attribute of None. The result of an expression will always have a memory space attribute associated with it. The unary logical negate operator, relational operators, and some functions return values that have a memory space attribute of N. The result of an expression that has only one operand (and possibly the unary negate or unary minus operator) always has the memory attribute of that
Chapter 3 Expressions - Expression Memory Space Attribute

Operands. Expressions that involve two or more operands and operators other than those mentioned above derive the memory space attribute of the result by examining the operands on the left and right side of an operator as shown in the following chart:

<table>
<thead>
<tr>
<th>Left Operand Memory Space Attribute</th>
<th>X</th>
<th>Y</th>
<th>L</th>
<th>P</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Right Operand Memory Space Attribute</td>
<td>X</td>
<td>X</td>
<td>*</td>
<td>X</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>*</td>
<td>Y</td>
<td>Y</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>L</td>
<td>X</td>
<td>Y</td>
<td>L</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>P</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>P</td>
</tr>
<tr>
<td></td>
<td>N</td>
<td>X</td>
<td>Y</td>
<td>L</td>
<td>P</td>
</tr>
</tbody>
</table>

* = Represents an illegal operation that will result in an error.

Notice that L memory space is regarded as a union of both X and Y space. In expressions that have one element that has a memory space attribute of L and another element with a memory space attribute of either X or Y, the result will have the more restrictive memory space attribute (X or Y).

The memory space attribute is regarded by the assembler as a type, in the same sense that high level languages use type for variables. Symbols that are assigned memory space attributes of X, Y, L, or P are assumed to be addresses and therefore can only have values between zero and the maximum address value of the DSP inclusive. Only symbols that have a memory space attribute of N can have values greater than the maximum address of the target processor.

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

- X:<address expression>
- Y:<address expression>
- L:<address expression>
- P:<address expression>

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

Expressions used for immediate addressing can have any memory space attribute.
3.4 Internal Expression Representation

Expression value representation internal to the assembler is dependent on the word size of the target processor. The assembler supports a word and a double word integer format internally. The actual storage size of an expression value is dependent upon the magnitude of the result, but the assembler is capable of representing signed integers up to 64 bits in length. These longer integer representations are useful when performing data initialization in L memory space.

Internal floating point representation is almost entirely dependent upon the host environment, but in general floating point values are stored in double precision format. This means that there are ordinarily 64 bits of storage allotted for a floating point number by the assembler, with 11 bits of exponent, 53 bits of mantissa, and an implied binary point.

3.5 Constants

Constants represent quantities of data that do not vary in value during the execution of a program.

3.5.1 Numeric Constants

Numeric constants can be in one of three bases:

**Binary**

Binary constants consist of a percent sign (%) followed by a string of binary digits (0,1).

Example: \(\%11010\)

**Hexadecimal**

Hexadecimal constants consist of a dollar sign ($) followed by a string of hexadecimal digits (0-9, A-F, a-f).

Example: \($12FF, $12ff\)

**Decimal**

Decimal constants can be either floating point or integer. Integer decimal constants consist of a string of decimal (0-9) digits optionally preceded by a grave accent (‘`). Floating point constants are indicated either by a preceding, following, or included decimal point or by the presence of an upper or lower case ‘E’ followed by the exponent.

Example:

\[12345 \quad \text{(integer)}\]
\[6E10 \quad \text{(floating point)}\]
\[.6 \quad \text{(floating point)}\]
\[2.7e2 \quad \text{(floating point)}\]
Chapter 3 Expressions - Operators

A constant may be written without a leading radix indicator if the input radix is changed using the RADIX directive. For example, a hexadecimal constant may be written without the leading dollar sign ($) if the input radix is set to 16 (assuming an initial radix of 10). The default radix is 10. See Chapter 6 on the RADIX directive for more information.

3.5.2 String Constants

String constants that are used in expressions are converted to a concatenated sequence of ASCII bytes (right aligned), as shown below. Strings used in expressions are limited to the long word size of the target processor; subsequent characters in the string are ignored. Null strings (strings that have no characters) have a value of 0.

String constants greater than the maximum number of characters can be used in expressions, but the assembler will truncate the value and will use only those characters that will fit in a DSP long word. In this case, a warning will be printed. This restriction also applies to string constants using the string concatenation operator. Handling of string constants by the DC and DCB directives is an exception to this rule; see Chapter 6 for a description.

Examples:

'ABC' $41424344
''$79 $00273739
'A' $00000041
'' $00000000 - null string
'abcdef' $61626364
'abc'++'de' $61626364

3.6 Operators

Some of the assembler operators can be used with both floating point and integer values. If one of the operands of the operator has a floating point value and the other has an integer value, the integer will be converted to a floating point value before the operator is applied and the result will be floating point. If both operands of the operator are integers, the result will be an integer value. Similarly, if both the operands are floating point, the result will be a floating point value.

3.6.1 Unary operators

plus (+)
minus (-)
one’s complement (~) - Integer only
logical negate (!)

The unary plus operator returns the value of its operand.
The unary minus operator returns the negative of its operand.
The one’s complement operator returns the one’s complement of its operand. It cannot be used with a floating point operand.

The unary logical negation operator returns an integer 1 (memory space attribute None) if the value of its operand is 0 and will return a 0 otherwise. For example, if the symbol BUF had a value of 0, then !BUF would have a value of 1. If BUF had a value of 1000, !BUF would have a value of 0.

### 3.6.2 Arithmetic operators

- **addition** (+)
- **subtraction** (-)
- **multiplication** (*)
- **division** (/)
- **mod** (%)

The addition operator yields the sum of its operands.

The subtraction operator yields the difference of its operands.

The multiplication operator yields the product of its operands.

The divide operator yields the quotient of the division of the first operand by the second. For integer operands the divide operation will produce a truncated integer result.

The mod operator applied to integers will yield the remainder from the division of the first operand by the second. If the mod operator is used with floating point operands, the mod operator will apply the following rules:

\[
Y \% Z = Y \quad \text{if } Z = 0
\]
\[
= X \quad \text{if } Z \not= 0
\]

where X has the same sign as Y, is less than Z, and satisfies the relationship:

\[
Y = i \times Z + X
\]

where i is an integer.

### 3.6.3 Shift operators

- **shift left** (<<) - Integer only
- **shift right** (>>) - Integer only

The shift left operator causes the left operand to be shifted to the left (and zero-filled) by the number of bits specified by the right operand.

The shift right operator causes the left operand to be shifted to the right by the number of bits specified by the right operand. The sign bit will be extended.
Shift operators cannot be applied to floating point operands.

**3.6.4 Relational operators**

- less than (`<`)
- less than or equal (`<=`)
- greater than (`>`)
- greater than or equal (`>=`)
- equal (`==`)
- not equal (`!=`)

Relational operators all work the same way. If the indicated condition is true, the result of the expression is an integer 1. If it is false, the result of the expression is an integer 0. In either case, the memory space attribute of the result is `None`.

For example, if D has a value of 3 and E has a value of 5, then the result of the expression `D < E` is 1, and the result of the expression `D > E` is 0. Each operand of the conditional operators can be either floating point or integer. Test for equality involving floating point values should be used with caution, since rounding error could cause unexpected results. Relational operators are primarily intended for use with the conditional assembly `IF` directive, but can be used in any expression.

**3.6.5 Bitwise operators**

- AND (`&`) - Integer only
- OR (`|`) - Integer only
- exclusive OR (`^`) - Integer only

The bitwise AND operator yields the bitwise AND function of its operands.

The bitwise OR operator yields the bitwise OR function of its operands.

The bitwise exclusive OR operator yields the bitwise exclusive OR function of its operands.

Bitwise operators cannot be applied to floating point operands.

**3.6.6 Logical operators**

- Logical AND (`&&`)
- Logical OR (`||`)

The logical AND operator returns an integer 1 if both of its operands are nonzero; otherwise, it returns an integer 0.
The logical OR operator returns an integer 1 if either of its operands is nonzero; otherwise it returns an integer 0.

The types of the operands may be either integer or floating point; the memory space attribute of the result is None. Logical operators are primarily intended for use with the conditional assembly IF directive, but can be used in any expression.

3.7 Operator Precedence

Expressions are evaluated with the following operator precedence:

1. parenthetical expression (innermost first)
2. unary plus, unary minus, one’s complement, logical negation
3. multiplication, division, mod
4. addition, subtraction
5. shift
6. relational operators: less, less or equal, greater, greater or equal
7. relational operators: equal, not equal
8. bitwise AND, OR, EOR
9. logical AND, OR

Operators of the same precedence are evaluated left to right. Valid operands include numeric constants, literal ASCII strings, and symbols. The one’s complement, shift, and bitwise operators cannot be applied to floating point operands. That is, if the evaluation of an expression (after operator precedence has been applied) results in a floating point number on either side of any of these operators, an error will be generated.

3.8 Functions

The assembler has several built-in functions to support data conversion, string comparison, and transcendental math computations. Functions may be used as terms in any arbitrary expression. Functions may have zero or more arguments, but must always be followed by open and closed parentheses. Function arguments which are expressions must be absolute expressions except where noted. Arguments containing external references are not allowed. There must be no intervening spaces between the function name and the opening parenthesis, and there must be no spaces between comma-separated arguments.

Assembler functions can be grouped into five types:

1. Mathematical functions
2. Conversion functions
3. String functions
4. Macro functions
5. Assembler mode functions
3.8.1 Mathematical Functions

The mathematical functions comprise transcendental, random value, and min/max functions, among others:

- ABS - Absolute value
- ACS - Arc cosine
- ASN - Arc sine
- AT2 - Arc tangent
- ATN - Arc tangent
- CEL - Ceiling function
- COH - Hyperbolic cosine
- COS - Cosine
- FLR - Floor function
- L10 - Log base 10
- LOG - Natural logarithm
- MAX - Maximum value
- MIN - Minimum value
- POW - Raise to a power
- RND - Random value
- SGN - Return sign
- SIN - Sine
- SNH - Hyperbolic sine
- SQT - Square root
- TAN - Tangent
- TNH - Hyperbolic tangent
- XPN - Exponential function
3.8.2 Conversion Functions

The conversion functions provide conversion between integer, floating point, and fixed point fractional values:

- **CVF** - Convert integer to floating point
- **CVI** - Convert floating point to integer
- **CVS** - Convert memory space
- **FLD** - Shift and mask operation
- **FRC** - Convert floating point to fractional
- **LFR** - Convert floating point to long fractional
- **LNG** - Concatenate to double word
- **LUN** - Convert long fractional to floating point
- **RVB** - Reverse bits in field
- **UNF** - Convert fractional to floating point

3.8.3 String Functions

String functions compare strings, return the length of a string, and return the position of a substring within a string:

- **LEN** - Length of string
- **POS** - Position of substring in string
- **SCP** - Compare strings

3.8.4 Macro Functions

Macro functions return information about macros:

- **ARG** - Macro argument function
- **CNT** - Macro argument count
- **MAC** - Macro definition function
- **MXP** - Macro expansion function
3.8.5 Assembler Mode Functions

Miscellaneous functions having to do with assembler operation:

- **CCC** - Cumulative cycle count
- **CHK** - Current instruction/data checksum
- **CTR** - Location counter type
- **DEF** - Symbol definition function
- **EXP** - Expression check
- **INT** - Integer check
- **LCV** - Location counter value
- **LST** - LIST directive flag value
- **MSP** - Memory space
- **REL** - Relative mode function

Individual descriptions of each of the assembler functions follow. They include usage guidelines, functional descriptions, and examples.

@@ABS(<expression>)@@

Returns the absolute value of <expression> as a floating point value. The memory space attribute of the result will be **None**.

Example:

```
MOVE #@@ABS(VAL),D4.S ; load absolute value
```

@@ACS(<expression>)@@

Returns the arc cosine of <expression> as a floating point value in the range zero to pi. The result of <expression> must be between -1 and 1. The memory space attribute of the result will be **None**.

Example:

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

@@ARG(<symbol> | <expression>)@@

Returns integer 1 if the macro argument represented by <symbol> or <expression> is present, 0 otherwise. If the argument is a symbol it must be single-quoted and refer to a dummy argument name. If the argument is an expression it refers to the ordinal position of the argument in the macro dummy argument list. A warning will be issued if this function is used when no macro expansion is active. The memory space attribute of the result will be **None**.

Example:

```
IF @@ARG(TWIDDLE) ; twiddle factor provided?
```
@ASN(<expression>)

Returns the arc sine of <expression> as a floating point value in the range -pi/2 to pi/2. The result of <expression> must be between -1 and 1. The memory space attribute of the result will be None.

Example:

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

@AT2(<expr1,expr2>)

Returns the arc tangent of <expr1>/<expr2> as a floating point value in the range -pi to pi. Expr1 and expr2 must be separated by a comma. The memory space attribute of the result will be None.

Example:

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

@ATN(<expression>)

Returns the arc tangent of <expression> as a floating point value in the range -pi/2 to pi/2. The memory space attribute of the result will be None.

Example:

MOVE # @ATN(1.0),D0.S ; load arc tangent

@CCC()

Returns the cumulative cycle count as an integer. Useful in conjunction with the CC, NOCC, and CONTCC assembler options (see the OPT directive). The memory space attribute of the result will be None.

Example:

IF @CCC() > 200 ; cycle count > 200?

@CEL(<expression>)

Returns a floating point value which represents the smallest integer greater than or equal to <expression>. The memory space attribute of the result will be None.

Example:

CEIL SET @CEL(-1.05) ; CEIL = -1.0
Chapter 3 Expressions - Functions

@CHK()

Returns the current instruction/data checksum value as an integer. Useful in conjunction with the CK, NOCK, and CONTCK assembler options (see the OPT directive). Note that assignment of the checksum value with directives other than SET could cause phasing errors due to different generated instruction values between passes. The memory space attribute of the result will be None.

Example:

    CHKSUM SET @CHK() ; reserve checksum value

@CNT()

Returns the count of the current macro expansion arguments as an integer. A warning will be issued if this function is used when no macro expansion is active. The memory space attribute of the result will be None.

Example:

    ARGCNT SET @CNT() ; squirrel away arg count

@COH(<expression>)

Returns the hyperbolic cosine of <expression> as a floating point value. The memory space attribute of the result will be None.

Example:

    HYCOS EQU @COH(VAL) ; compute hyperbolic cosine

@COS(<expression>)

Returns the cosine of <expression> as a floating point value. The memory space attribute of the result will be None.

Example:

    DC -@COS(@CVF(COUNT)*FREQ) ; compute cosine value

@CTR([L | R])

If L is specified as the argument, returns the counter number of the load location counter. If R is specified, returns the counter number of the runtime location counter. The counter number is returned as an integer value with memory space of None.

Example:

    CNUM = @CTR(R) ; runtime counter number
@CVF(<expression>)

Converts the result of <expression> to a floating point value. The memory space attribute of the result will be None.

Example:

    FLOAT SET @CVF(5) ; FLOAT = 5.0

@CVI(<expression>)

Converts the result of <expression> to an integer value. This function should be used with caution since the conversions can be inexact (e.g., floating point values are truncated). The memory space attribute of the result will be None.

Example:

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

@CVS({X | Y | L | P | N},<expression>)

Converts the memory space attribute of <expression> to that specified by the first argument; returns <expression>. See section 3.3 for more information on memory space attributes. The <expression> may be relative or absolute.

Example:

    LOADDR EQU @CVS(X,TARGET) ; set LOADDR to X:TARGET

@DEF(<symbol>)

Returns an integer 1 (memory space attribute N) if <symbol> has been defined, 0 otherwise. <symbol> may be any label not associated with a MACRO or SECTION directive. If <symbol> is quoted it is looked up as a DEFINE symbol; if it is not quoted it is looked up as an ordinary label.

Example:

    IF @DEF(ANGLE) ; assemble if ANGLE defined

@EXP(<expression>)

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

Example:

    IF !@EXP(@FRC(VAL)) ; skip on error
Chapter 3 Expressions - Functions

@FLD(<base>,<value>,<width>[,<start>])

Shift and mask <value> into <base> for <width> bits beginning at bit <start>. If <start> is omitted, zero (least significant bit) is assumed. All arguments must be positive integers and none may be greater than the target word size. Returns the shifted and masked value with a memory space attribute of None.

Example:

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

@FLR(<expression>)

Returns a floating point value which represents the largest integer less than or equal to <expression>. The memory space attribute of the result will be None.

Example:

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

@FRC(<expression>)

For binary fractional DSPs (DSP56300) this function performs scaling and convergent rounding to obtain the fractional representation of the floating point <expression> as an integer. The memory space attribute of the result will be None.

Example:

```
FRAC EQU @FRC(FLT)+1 ; compute saturation
```

@INT(<expression>)

Returns an integer 1 (memory space attribute N) if <expression> has an integer result, 0 otherwise. The <expression> may be relative or absolute.

Example:

```
IF @INT(TERM) ; insure integer value
```

@L10(<expression>)

Returns the base 10 logarithm of <expression> as a floating point value. <expression> must be greater than zero. The memory space attribute of the result will be None.

Example:

```
LOG EQU @L10(100.0) ; LOG = 2
```
@LCV([L | R], [L | H | <expression>])

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

The @LCV function will not work correctly if used to specify the runtime counter value of a relocatable overlay. This is because the resulting value is an overlay expression, and overlay expressions may not be used to set the runtime counter for a subsequent overlay. See the ORG directive (Chapter 6) for more information.

Example:

ADDR = @LCV(R) ; save runtime address

@LEN(<string>)

Returns the length of <string> as an integer. The memory space attribute of the result will be None.

Example:

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

@LFR(<expression>)

For binary fractional DSPs (DSP56300) this function performs scaling and convergent rounding to obtain the fractional representation of the floating point <expression> as a long integer. The memory space attribute of the result will be None.

Example:

LFRAC EQU @LFR(LFLT) ; store binary form

@LNG(<expr1>,<expr2>)

Concatenates the single word <expr1> and <expr2> into a double word value such that <expr1> is the high word and <expr2> is the low word. The memory space attribute of the result will be None.

Example:

LWORD DC @LNG(HI,LO) ; build long word
Chapter 3 Expressions - Functions

@LOG(<expression>)

Returns the natural logarithm of <expression> as a floating point value. <expression> must be greater than zero. The memory space attribute of the result will be None.

Example:

LOG EQU @LOG(100.0) ; LOG = 4.605170

@LST()

Returns the value of the LIST directive flag as an integer, with a memory space attribute of None. Whenever a LIST directive is encountered in the assembler source, the flag is incremented; when a Nolist directive is encountered, the flag is decremented.

Example:

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

@LUN(<expression>)

Converts the double-word <expression> to a floating point value. For fractional DSPs (DSP56300) <expression> should represent a binary fraction. The memory space attribute of the result will be None.

Example:

DBLFRC EQU @LUN($3FE0000000000000) ;DBLFRC = 0.5

@MAC(<symbol>)

Returns an integer 1 (memory space attribute N) if <symbol> has been defined as a macro name, 0 otherwise.

Example:

IF @MAC(DOMUL) ; expand macro

@MAX(<expr1>[,...,<exprN>])

Returns the greatest of <expr1>,...,<exprN> as a floating point value. The memory space attribute of the result will be None.

Example:

MAX DC @MAX(1.0,5.5,-3.25) ; MAX = 5.5
\texttt{@MIN(<expr1>,...,<exprN>)}

Returns the least of <expr1>,...,<exprN> as a floating point value. The memory space attribute of the result will be \texttt{None}.

Example:

\begin{verbatim}
MIN DC @MIN(1.0,5.5,-3.25) ; MIN = -3.25
\end{verbatim}

\texttt{@MSP(<expression>)}

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

- None = 0
- X space = 1
- Y space = 2
- L space = 3
- P space = 4

The <expression> may be relative or absolute.

Example:

\begin{verbatim}
MEM SET @MSP(ORIGIN) ; save memory space
\end{verbatim}

\texttt{@MXP()}

Returns an integer 1 (memory space attribute \texttt{N}) if the assembler is expanding a macro, 0 otherwise.

Example:

\begin{verbatim}
IF @MXP() ; macro expansion active?
\end{verbatim}

\texttt{@POS(<str1>,<str2>[,<start>])}

Returns the position of string <str2> in <str1> as an integer, starting at position <start>. If <start> is not given the search begins at the beginning of <str1>. If the <start> argument is specified it must be a positive integer and cannot exceed the length of the source string. The memory space attribute of the result will be \texttt{None}.

Example:

\begin{verbatim}
ID EQU @POS('DSP56300','56') ; ID = 3
\end{verbatim}
Chapter 3 Expressions - Functions

@POW(<expr1>,<expr2>)

Returns <expr1> raised to the power <expr2> as a floating point value. <expr1> and <expr2> must be separated by a comma. The memory space attribute of the result will be None.

Example:

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

@REL()

Returns an integer 1 (memory space attribute N) if the assembler is operating in relative mode, 0 otherwise.

Example:

IF @REL(); in relative mode?

@RND()

Returns a random value in the range 0.0 to 1.0. The memory space attribute of the result will be None.

Example:

SEED DC @RND(); save initial seed value

@RVB(<expr1>[,<expr2>])

Reverse the bits in <expr1> delimited by the number of bits in <expr2>. If <expr2> is omitted the field is bounded by the target word size. Both expressions must be single word integer values.

Example:

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

@SCP(<str1>,<str2>)

Returns an integer 1 (memory space attribute N) if the two strings compare, 0 otherwise. The two strings must be separated by a comma.

Example:

IF @SCP(STR,'MAIN'); does STR equal MAIN?
@SGN(<expression>)

Returns the sign of <expression> as an integer: -1 if the argument is negative, 0 if zero, 1 if positive. The memory space attribute of the result will be None. The <expression> may be relative or absolute.

Example:

IF @SGN(INPUT) ; is sign positive?

@SIN(<expression>)

Returns the sine of <expression> as a floating point value. The memory space attribute of the result will be None.

Example:

DC @SIN(@CVF(COUNT)*FREQ) ; compute sine value

@SNH(<expression>)

Returns the hyperbolic sine of <expression> as a floating point value. The memory space attribute of the result will be None.

Example:

HSINE EQU @SNH(VAL) ; hyperbolic sine

@SQT(<expression>)

Returns the square root of <expression> as a floating point value. <expression> must be positive. The memory space attribute of the result will be None.

Example:

SQRT EQU @SQT(3.5) ; SQRT = 1.870829

@TAN(<expression>)

Returns the tangent of <expression> as a floating point value. The memory space attribute of the result will be None.

Example:

MOVE #@TAN(1.0),D1.S ; load tangent
Chapter 3 Expressions - Functions

@TNH(<expression>)

Returns the hyperbolic tangent of <expression> as a floating point value. The memory space attribute of the result will be None.

Example:

HTAN = @TNH(VAL) ; hyperbolic tangent

@UNF(<expression>)

Converts <expression> to a floating point value. For fractional DSPs (DSP56300) <expression> should represent a binary fraction. The memory space attribute of the result will be None.

Example:

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

@XPN(<expression>)

Returns the exponential function (base e raised to the power of <expression>) as a floating point value. The memory space attribute of the result will be None.

Example:

EXP EQU @XPN(1.0) ; EXP = 2.718282
Chapter 4 Software Project Management

4.1 Introduction

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

4.2 Sections

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

```
SECTION <section name> [GLOBAL I STATIC I LOCAL]

Section source statements

ENDSEC
```

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

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

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

```
SYM1 EQU 1
SYM2 EQU 2

SECTION EXAMPLE

SYM1 EQU 3
MOVE #SYM1,R0
MOVE #SYM2,R1

ENDSEC

MOVE #SYM1,R2
```

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

4.3.1 Sections and Symbols

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

```
XDEF <symbol>,<symbol>,...,<symbol>
```

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

```
XREF <symbol>,<symbol>,...,<symbol>
```
XDEFed symbols by default are recognized only in other sections which XREF them. They can be made fully global (recognizable by sections which do not XREF them) by use of the XR option (see the OPT directive, Chapter 6). Alternatively the GLOBAL directive (see Chapter 6) may be used within a section to make the named symbols visible outside of the section. Both the XDEF and XREF directives must be used before the symbols to which they refer are defined or used in the section. Here is another example:

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

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

If the GLOBAL qualifier follows the <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 the GLOBAL directive. This is useful when a section needs to be independently relocatable, but data hiding is not required.

If the LOCAL qualifier follows the <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.

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

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

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

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

4.3.3 Nested and Fragmented Sections

Sections can be nested to any level. When the assembler encounters a nested section, the current section is stacked and the new section is used. When the ENDSEC directive of the nested section is encountered, the assembler restores the old section and uses it. The ENDSEC directive always applies to the most recent SECTION directive. Nesting sections provides a measure of scoping for symbol names, in that symbols defined within a given section are visible to other sections nested within it. For example, if section B is nested inside section A, then a symbol defined in section A can be used in section B without XDEFing in section A or XREFing in section B. This scoping behavior can be turned off and on with the NONS and NS options respectively (see the OPT directive, Chapter 6).

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.
4.4 Sections and Relocation

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. 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.

If the assembler encounters an ORG directive with an absolute runtime address specification it switches to absolute mode and begins generating absolute addresses within the enclosing section. Note that the mode change is effective only if the assembler was started in relative mode; if the -A command line option is used the assembler always generates absolute addresses. The assembler continues to generate absolute code until an ENDSEC directive is encountered, or the mode is explicitly changed via the MODE directive.

The MODE directive allows for arbitrary switching between absolute and relocatable code generation:

\[
\text{MODE} \quad \langle \text{ABSOLUTE} \mid \text{RELATIVE} \rangle
\]

The MODE directive may be issued 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 within a section are combined at link time, even if absolute blocks are inter-
spersed among relocatable blocks. The **MODE** directive has no effect when the command line -A option is active. The following is an example:

```
SECTION EXAMPLE ; relocatable section

; code/data generated here is relocatable

MODE ABSOLUTE

; code/data generated here is absolute; it will be
; placed in memory at the location specified during
; assembly

MODE REL

; back to relocatable; code/data generated here
; will be combined with the previous relocatable block,
; as long as memory space and mappings are compatible

ORG P:$200

; code/data generated here will be absolute
; until ENDSEC directive is found

ENDSEC
```

More information on the **ORG** and **MODE** directives can be found in Address Assignment and under the individual directive descriptions in Chapter 6.

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.

### 4.5 Address Assignment

The Freescale DSP assembler can support absolute address assignment at assembly time or generation of relocatable program addresses which are resolved during the linking phase. The **ORG** directive is used to specify memory space changes, mappings to physical memory, and absolute address assignment.
Various memory layouts require special handling for data generation or location counter updating. In the case of \texttt{L} memory, two words of code or data are produced for each increment of the location counter.

The assembler allows for two sets of program counters per memory space, a set of load counters and a set of runtime counters. The distinction between load and runtime counters is maintained so that the assembler can support \textit{overlays}, or runtime transfers of code/data from one memory space to another. In these cases code or data might be loaded in one memory space at a given address, but then copied to a different memory space and address for execution. The assembler can produce output for either absolute or relocatable overlays.

Freescale DSPs are capable of performing special-purpose addressing on data structures suited to digital signal processing applications. Two such data structures are the modulo buffer and the reverse-carry buffer, collectively referred to as \textit{circular buffers}. Due to the way they are accessed and manipulated, these buffers generally are constrained to a particular size or starting address. The assembler provides directives for aligning buffer base addresses, allocating buffer space, and initializing buffer contents.

4.5.1 \textbf{The ORG Directive}

The \texttt{ORG} directive specifies which memory space will be the runtime memory space and which counter (the \texttt{H}, \texttt{L}, default, or numbered runtime counter associated with that memory space and section) will be the runtime location counter. At the same time, the \texttt{ORG} directive indicates which memory space will be the load memory space and which counter (the \texttt{H}, \texttt{L}, default, or numbered load counter associated with that memory space and section) will be used as the load location counter. In addition, the \texttt{ORG} directive can be used to specify a physical mapping to DSP memory and to assign initial values to the runtime and load location counters.

The names of the counters (\texttt{High}, \texttt{Low}, and default) are symbolic only, and the assembler performs no checks to insure that the value assigned to the \texttt{High} counter is greater than the \texttt{Low}. Moreover, there is no inherent relationship among numbered counters, except that counters 0, 1, and 2 correspond to the default, \texttt{Low}, and \texttt{High} counters, respectively. Counters are useful for providing mnemonic links between runtime and load memory spaces or among individual memory blocks. Separate counters can be used to obtain blocks within a common section which are accessed from one memory space but mapped to separate physical memories. Also counters are necessary for handling relocatable overlays at link time, as the DSP linker does not support the notion of separate load and runtime counters. See the examples below for more information on location counter usage.

The \texttt{ORG} directive is organized as follows:

\begin{verbatim}
ORG <rms>[<rlc>]<rmp>:[<exp1>],<lms>[<llc>]<lmp>:[<exp2>]]
\end{verbatim}
or alternatively:

\[ \text{ORG} \ <\text{rms}>[\langle\text{rmp}\rangle][\langle\text{rce}\rangle];[\langle\text{exp1}\rangle],[\langle\text{lms}\rangle[\langle\text{lmp}\rangle][\langle\text{lce}\rangle];[\langle\text{exp2}\rangle]] \]

\(<\text{rms}>\)

Which memory space (X, Y, L, or P) 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.

\(<\text{rlc}>\)

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.

\(<\text{rmp}>\)

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.

\(<\text{rce}>\)

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.

\(<\text{exp1}>\)

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.

\(<\text{lms}>\)

Which memory space (X, Y, L, or P) 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.

\(<\text{lmc}>\)

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.
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.

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.

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.

The **ORG** directive is useful in multi-programmer projects because it provides a means for the individual programmer to specify in which memory space and which segment of that memory space the code being written will be located without specifying an absolute address. Absolute address assignment can be deferred until the various components of the program are brought together. The utility of the **ORG** directive is not limited to multi-programmer projects. Even in single programmer projects, the **ORG** directive supports manipulation of overlays and the intermixing of label definition and code generation in multiple memory spaces without having to reinitialize a location counter every time the load memory space is changed.

**4.5.2 Overlays**

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 section 1.6 for more information on location counters and overlays.
Chapter 4 Software Project Management - Address Assignment

The values and memory space attributes of both the load and runtime location counters can be accessed with the @LCV function (see Section 3.8). This is particularly useful when assigning the load location counter value to a label as a reference point for the overlay manager part of the program. The High, Low, default, or numbered counter assignment can be determined by using the @CTR function (Section 3.8).

4.5.3 Address Assignment Examples

Some examples of the ORG directive are as follows:

**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** P: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 byte-wise 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.

If the last example shown was used in the following code sequence (assume the runtime counter associated with P space had a previous value of $0010),

```
ORG P(5):,Y:$8000
RLMUL MOVE X:(R0),D4.S Y:(R4),D7.S
FMPY.S D4,D7,D0
MOVE D0.S,X:(R1)
```

then the label RLMUL would have a value of $0010 and a memory space attribute of P space; the code generated would load into Y memory starting at $8000; and the runtime address associated with the code would start at $0010.

### 4.5.4 Circular Buffers

To take advantage of the special DSP addressing capabilities a circular buffer must be aligned on an appropriate address boundary with respect to its size. For a buffer to be located properly in memory the lower bits of the starting address which encompass one less than the buffer size must be zero. For example, the lowest address greater than zero at which a buffer of size 32 may be located is 32 (20 hexadecimal). More generally, the buffer base address must be modulo the buffer size, or a multiple of $2^k$, where $2^k$ is greater than or equal to the size of the buffer.

Buffers may be allocated manually or by using one of the assembler’s special buffer directives:

```
ORG X:$100
BUF1 DS 24
BUF2 DSM 32
```

The ORG statement sets the origin to hexadecimal 100 in X memory. The first buffer BUF1 is manually allocated with a size of 24. Since the starting address is hex 100 the buffer is already suitably aligned. The label BUF1 is assigned the runtime counter value at the beginning of the buffer. The second buffer is allocated using the DSM directive, which automatically sets the buffer starting address before reserving space. In this case, the first buffer ended at location 117 hexadecimal, so the assembler advances the program counter to location 120 hex before assigning a value to the buffer label BUF2.
Buffers are special-purpose data structures, but they are named and accessed with labels like any other data block. They therefore adhere to the same rules governing data hiding in sections that any other segment of code or data would follow. A buffer allocated when the assembler is in absolute mode (either via -A or an absolute ORG or MODE directive) is placed in memory according to the absolute value of the runtime location counter at assembly time. A buffer allocated in relative mode (a relocatable buffer) is suitably aligned within its relocation section at assembly time. During the link phase a section enclosing any relocatable buffers is located based on the largest relocatable buffer it contains, unless the buffers inside the section are auto-aligned. This insures that any smaller buffers within the section are properly aligned. If any buffers in the section are auto-aligned, they will be relocated independent of any other code or data in the section. Note that repositioning of any buffer, whether relocatable or absolute, may result in alignment gaps in memory depending on the layout of data surrounding the buffer blocks.

4.6 Example 1: Multi-Programmer Environment

Typical multi-programmer projects are often split into tasks representing functional units. For discussion purposes, suppose a project has been divided into three tasks - I/O, FILTER, and MAIN. Each task will be written by a separate programmer as a separate section. For example, when the I/O task has been written, there will be a file called IO.ASM. This file will have the following form:

```asm
SECTION I_O
XREF I_PORT,O_PORT
ORG XL:
.
.
<storage location definitions>
.
.
ORG P:
.
.
<IO section source statements>
.
.
ENDSEC
```

In this example, because the X space storage locations were defined within the section I_O, they will be private storage locations that are accessible only by the I_O handler, and cannot be referenced by other sections. If global memory resource management is desired, then the I_O section would not have defined any storage locations, and these would have been defined as XREF. The X space data will be addressed through the Low
runtime counter. The P memory code is also private to the I_O section and uses the default runtime location counter for address generation.

In the discussion below, assume that the programmers responsible for the FILTER and MAIN sections have similar program structures located in files named FILTER.ASM and MAIN.ASM respectively. The program units can be combined either by invoking a final assembly step to assign absolute addresses, or by assembling the modules separately and then linking.

### 4.6.1 Absolute Mode Implementation

To assemble the entire project source code, a new file called PROJECT.ASM would be created and would have the form:

```assembly
ORG XLE:$0000

<global low memory X storage declarations (if any)>

ORG YLE:$0000

<global low memory Y storage declarations (if any)>

ORG YH:$FFC0

<global high memory Y storage declarations (if any)>

ORG XH:$FFC0

<global high memory X storage declarations (if any)>

; initialize internal low Program memory location counter
ORG PL:$1000

; initialize external high Program memory location counter
ORG PHE:$F000

INCLUDE 'MAIN.ASM'
INCLUDE 'IO.ASM'
INCLUDE 'FILTER.ASM'

END ENTRY
```

This file provides the project manager with a mechanism to organize memory utilization to suit the application. For example, the external high P memory initialization statement might correspond to the memory location of an external EPROM.
After the location counters corresponding to the X, Y, and P(rogram) memory spaces are initialized, the assembler is directed to take input from the MAIN.ASM file with the INCLUDE directive. Within the MAIN.ASM file, the source statements are assembled and object code is generated. The X, Y, L, and P(rogram) location counters (High, Low) are advanced corresponding to the number of words generated for each memory space and location counter in use.

When the end of the MAIN.ASM file is encountered, the assembler returns to the next sequential statement in the PROJECT.ASM file. This directs the assembler to start taking input from the IO.ASM file. Within this file, the ORG PL: statement directs the assembler to set the current memory space to P(rogram) and restore the last used P(ogram) L(ow) location counter. The <IO source program statements> shown previously will be assembled at the next available Low Program memory space. When the end of the IO.ASM file is encountered, the X, Y, and P(rogram) location counters (High and Low) will have been advanced corresponding to the number of words generated for each memory space.

In a similar manner, the file FILTER.ASM will be assembled. The last statement of the PROJECT.ASM file informs the assembler that this is the last logical source statement, and the starting address for the object module will be a label called ENTRY. In the example above, ENTRY would have been a label defined in the section MAIN and declared as global with the XDEF directive.

### 4.6.2 Relative Mode Implementation

Using the assembler default relative mode, each of the source files is assembled separately. For each section defined in the input files a separate set of location counters is maintained such that all memory spaces for each section begin at relative address zero. The linker is invoked to combine the files and establish base addresses:

```
DSPLNK -B -M -OXLI:0 -OYLI:0 -OYH:FFC0 -OXH:FFC0 -OPL:1000 -OPHE:A000 MAIN IO FILTER
```

The linker reads the command input and sets up base values for all counters specified on the command line. In this example, the X and Y low memory counters are initialized to zero, whereas the X and Y high memory counters are set to FFC0 hexadecimal. The program low and high memory counters are initialized similarly. When the linker creates the executable file it reads the input files and sets the starting address for all sections relative to the values obtained from the command line. As the MAIN object file is read the linker increments the section counters for all appropriate memory spaces.

After the MAIN object file is processed, the IO object file is read. The section named I_O contained an ORG directive indicating a switch to the low X data memory counter. Recall that the assembler generated relocatable code for the I_O section source such that the low X data memory counter begins at zero. The linker adjusts the low X memory counter associated with section I_O to reflect any previous data generation performed in low X memory (e.g. in MAIN). The FILTER module is linked in a similar fashion.
Another way for specifying base addresses, instead of lengthy command line options, is through a memory control file. The memory control file allows the programmer to indicate memory space starting addresses analogously to the command line approach. In addition, the memory control file offers finer control over placement of sections in memory. See the Freescale DSP Linker/Librarian Reference Manual for more information on the memory control file.

The preceding examples described two methods for organizing a software project. Refer to the descriptions of the ORG and SECTION directives in Chapter 6 for a more detailed discussion. See also the Freescale DSP Linker/Librarian Reference Manual for more information on relocation and linking. One other assembler directive that should be mentioned (although not shown in the previous example) is the MACLIB directive which allows sections to share a common macro library. The MACLIB directive is discussed more fully in Chapter 5.

### 4.7 Example 2: Overlays

An overlay is a transfer of code or data from one memory space or address to another memory space or address at runtime. Often the transfer involves copying different blocks of code or data over a common storage area as runtime circumstances dictate; hence the name overlay. Overlays are useful for moving code into internal program memory from an external memory source such as EPROM. They are also effective when implementing large programs with multiple segments which do not need to be accessed concurrently. Consider the following program fragment contained in a file called OVER1.ASM:

```
SECTION OVERLAY1
XREF OVLBASE
XDEF OVL1,O1SIZE
ORG X:
  OVL1
  ORG P:OVLBASE,X:
START .
  .
  <overlay source statements>
  .
END O1SIZE EQU END-START
ENDSEC
```

This is a sample of overlay code bounded by a SECTION directive. The overlay base, or the place to which this block of code will be moved for execution, is declared external at OVLBASE (OVLBASE is actually defined elsewhere). The label OVL1 is XDEFed to provide a handle for moving the block at runtime, and O1SIZE is also XDEFed so that the overlay management code knows how many words to move. Note that the OVL1 label is placed before the ORG for the overlay so that it remains a valid address in X memory dur-
Chapter 4 Software Project Management - Example 2: Overlays

ing execution. The overlay \texttt{ORG} directive insures that subsequent addresses will be based from OVLBASE at runtime. The size of the overlay block (O1SIZE) is computed by subtracting the START label value from the END label address. Assume for purposes of discussion that there are other files containing similar overlay code with names OVER2.ASM and OVER3.ASM.

4.7.1 Absolute Mode Implementation

In order to avoid binding addresses within the individual overlay modules, the programmer could devise a preamble file called OVLPROJ.ASM which sets the appropriate counters and establishes the overlay base address.

<table>
<thead>
<tr>
<th>SECTION</th>
<th>OVLPROJECT</th>
</tr>
</thead>
<tbody>
<tr>
<td>XDEF</td>
<td>OVLBASE</td>
</tr>
<tr>
<td>XREF</td>
<td>OVL1,O1SIZE</td>
</tr>
<tr>
<td>XREF</td>
<td>OVL2,O2SIZE</td>
</tr>
<tr>
<td>XREF</td>
<td>OVL3,O3SIZE</td>
</tr>
<tr>
<td>ORG</td>
<td>XE:$100    ; set absolute base for overlay sections</td>
</tr>
<tr>
<td>ORG</td>
<td>PI:$200    ; set absolute base address for overlay</td>
</tr>
<tr>
<td>OVLBASE</td>
<td>DS $400    ; reserve space for overlay area</td>
</tr>
<tr>
<td>MOVEOV1</td>
<td>; code to move first overlay segment</td>
</tr>
<tr>
<td>MOVE</td>
<td>#OVL1,R0   ; load overlay code address</td>
</tr>
<tr>
<td>MOVE</td>
<td>#OVLBASE,R1; load overlay base address</td>
</tr>
<tr>
<td>MOVE</td>
<td>#O1SIZE,R2 ; load overlay code size</td>
</tr>
<tr>
<td>DO</td>
<td>R2,_ENDLOOP; loop to move data words into P memory</td>
</tr>
<tr>
<td>MOVE</td>
<td>X:(R0)+,X0 ; get word of overlay from data memory</td>
</tr>
<tr>
<td>MOVE</td>
<td>X0,P:(R1)+ ; store word of overlay into P memory</td>
</tr>
</tbody>
</table>

_\texttt{ENDSEC}_

The overlay base address OVLBASE is made global with the \texttt{XDEF} statement. The overlay segments and their sizes are made visible to the project section by using the \texttt{XREF} directive. The first \texttt{ORG} establishes where the overlay segments will be placed in memory contiguously at load time. The second \texttt{ORG} sets up the absolute base address for the overlay area common to all of the overlay segments. Uninitialized space is allocated for the overlay area, immediately followed by code to move the overlay segments into the common area at runtime. The following assembler command line will process the header file and all overlay segments:

\texttt{ASM56300 -A -B -L OVLPROJ OVER1 OVER2 OVER3 START}
The assembler is invoked in absolute mode (-A option), and generates an executable and listing file. All files on the command line are processed as a single assembly run and all are used to produce the output. OVLPROJ.ASM is read first and sets up the appropriate absolute addresses for later sections. Then each overlay file is read and loaded one after the other at external X memory address 100 hexadecimal. However, since each overlay module was intended to run starting at OVLBASE in P memory, all labels and jumps to those labels within the overlay code will be relative to the overlay base address. This means that the code in each of the overlay modules, when loaded by the overlay management code in the OVLPROJECT section, will start executing at internal P memory address 200 hexadecimal. The file START.ASM contains an **END** directive which indicates the program start address after loading.

### 4.7.2 Relative Mode Implementation

In relative mode each of the overlay files is assembled separately to create individual object files. The object files are combined to build a single executable file. A preamble file OVLPROJ.ASM containing overlay management code might appear as follows.

```
SECTION OVLPROJECT
XDEF OVLBASE
XREF OVL1,O1SIZE
XREF OVL2,O2SIZE
XREF OVL3,O3SIZE
ORG PI: ; set base address for overlay
MOVE OV1 ; code to move first overlay segment
  MOVE #OVL1,R0 ; load overlay code address
  MOVE #OVLBASE,R1 ; load overlay base address
  MOVE #O1SIZE,R2 ; load overlay code size
  DO R2,_ENDLOOP ; loop to move data words into P memory
  MOVE X:(R0)+,X0 ; get word of overlay from data memory
  MOVE X0,P:(R1)+ ; store one word of overlay into P memory

_ENDLOOP
.
.
.
OVLBASE DS $400 ; reserve space for overlay area
ENDSEC START
```

Note that the **ORG** to P space does not specify an absolute address. In order to obtain the same result from these files as in an absolute mode implementation the following linker command line would be used:

```
DSPLNK -B -M -OXE:100 -OPI:200 OVLPROJ OVER1 OVER2 OVER3
```
The linker scans the command line and sets the base addresses for X and P memory. Here the X default counter is set to hex 100 and mapped to external memory; likewise the P default counter is set to hex 200 and mapped to internal memory. Base addresses can also be established with the linker memory control file.

The linker reads each input object file, placing the header file in internal P memory and combining the overlay modules into a contiguous block loaded into external X memory at location 100 hexadecimal. Any labels or jumps within the overlay blocks are resolved to addresses relative to the relocatable symbol OVLBASE. Since OVLBASE is the first load P memory address it is assigned the value 200 hexadecimal. The linker does not guarantee that a given symbol or section will begin at a particular location unless that information is explicitly specified in the linker memory control file. For more information on specific linker operations see the Freescale DSP Linker/Librarian Reference Manual.

4.8 Example 3: Bootstrap Overlay

Many Freescale DSP processors, specifically those with RAM-based program memory, support a bootstrap mode of operation. This involves mapping a built-in ROM-based bootstrap program into P memory, executing the program to move user-supplied code from another location (usually EPROM) into program RAM, then transferring control to the user program. Because the user program is loaded in one location (e.g. EPROM) but moved to another for execution, it is a natural application for assembly language overlay semantics. Another wrinkle in bootstrap mode is that user instruction words are loaded in byte-wise fashion, such that the load location counter must be incremented by bytes rather
than words. Consider the following section fragments contained in two files called SECT1.ASM and SECT2.ASM respectively:

```
SECTION SECT1
ORG PI(1):,PE(2)
START1
  ...
  <source statements>
  ...
END1
ENDSEC

SECTION SECT2
ORG PI(1):,PE(2)
START2
  ...
  <source statements>
  ...
END2
ENDSEC
```

4.8.1 Absolute Mode Implementation

In order to avoid binding addresses within the individual modules, the programmer could devise a preamble file called BOOTPROJ.ASM which sets the appropriate options and establishes load and runtime base addresses:

```
OPT LB ; increment load counter by bytes
ORG PI(1):$100,PE(2):$C000 ; set runtime RAM address, bytewide load ROM address
```

The OPT directive with the LB option indicates that the assembler should increment the load counter by the number of bytes in the target processor word. This guarantees that the EPROM addresses will be correct for bytewide loading during bootstrap processing. In the ORG directive, the runtime location counter, tagged as 1 and mapped to internal memory, is set to hex 100. The load counter is tagged as 2, mapped to external memory, and set to hex C000, where the built-in bootstrap program will begin loading bytes after processor reset. The files are assembled using the command below:

```
ASM56300 -A -B -L BOOTPROJ SECT1 SECT2
```
Chapter 4 Software Project Management - Example 3: Bootstrap Overlay

The assembler is invoked in absolute mode (-A option), and generates an executable and listing file. All files on the command line are processed as a single assembly run and all are used to produce the output. BOOTPROJ.ASM is read first and sets up the appropriate absolute addresses for later sections. Since no explicit base address was given in the section files, both load and runtime addresses will continue from one section to the other, e.g. they will be contiguous. For example, if only two words of instruction were between each of the START and END labels, the runtime value for END1 and START2 would be hex 102. However, the load address of the code associated with these labels, assuming a 16 bit target word size, would be C004 hexadecimal. Similarly, the runtime value for END2 would be 104 hex and the corresponding load address would be C008 hexadecimal.

4.8.2 Relative Mode Implementation

In relative mode each of the source files is assembled separately to create individual object files. The object files are combined to build a single executable file. A preamble file is not necessary to handle bootstrap files in relative mode because the addresses are established at link time. In order to generate byte-wide load addresses the LB option can be specified on the assembler command line using the -O command line option:

```
ASM56300 -B -L -OLB SECT1
```

This command assembles the file SECT1.ASM and creates a relocatable object file called SECT1.CLN. The listing file shows that the starting address of the section is zero; however, because of the LB option on the command line the load counter will increment at three times the rate of the runtime counter (assuming a 24-bit DSP56300 family target processor). A similar command is used to assemble the SECT2 module. The two files are linked as follows:

```
DSPLNK -BBBOOTPROJ.CLD "-OPI(1):100" "-OPE(2):C000" SECT1 SECT2
```

The linker scans the command line and sets the base addresses for P internal and external memory. The quotes around the -O options are necessary to avoid interpretation of parentheses by some host command interpreters. Here the P counter number 1 is set to hex 100 and mapped to internal memory; likewise the P counter number 2 is set to hex C000 and mapped to external memory. Base addresses can also be established with the linker memory control file. Since no explicit overlay base addresses were encountered in the source files, both load and runtime addresses for the sections will be adjacent and non-overlapping. Assuming a code size of 2 for each section and a 16 bit word size, the value for label START1 will be hex 100 and the value for START2 will be hex 102; the corresponding load addresses will be C000 hex and C004 hex, respectively. The executable output will be written to the file BOOTPROJ.CLD.
Chapter 5 Macro Operations and Conditional Assembly

5.1 Macro Operations

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

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

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

To invoke a macro, the macro name must appear in the operation code field of a source statement. Any arguments are placed in the operand field. By suitably selecting the arguments in relation to their use as indicated by the macro definition, the programmer causes the assembler to produce in-line coding variations of the macro definition.
Chapter 5 Macro Operations and Conditional Assembly - Macro Libraries

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

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

5.2 Macro Libraries

The Freescale DSP assembler allows for the maintenance of macro libraries with the MACLIB directive. This directive is used to specify the pathname (as defined by the host 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 a MACLIB directive has been specified in the source code and the assembler encounters a name in the operation field that is not a previously defined macro or is not contained in the directive or mnemonic tables, the directory specified in the MACLIB directive will be searched for a file of that 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 name, or 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 they were specified.

5.3 Macro Definition

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

The header of a macro definition has the form:

```
<label> MACRO <dummy argument list> <comment>
```
Chapter 5 Macro Operations and Conditional Assembly - Macro Definition

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

\[ [<\text{dumarg}>[,<\text{dumarg}>,...,<\text{dumarg}>]] \]

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 global symbol names. Dummy argument names that are preceded by an underscore are not allowed. Dummy arguments are separated by commas.

For example, consider the following macro definition:

```
N_R_MUL MACRO NMUL,AVEC,BVEC,RESULT
    header
    ;This macro implements N real multiplies
    ;RESULT(I) = AVEC(I) * BVEC(I) I=1..NMUL
    ;where
    ; NMUL = number of multiplications
    ; AVEC = base address of array AVEC(I)
    ; BVEC = base address of array BVEC(I)
    ; RESULT = base address of array RESULT(I)
    ;
    MOV #AVEC,R0
    MOV #BVEC,R4
    MOV #RESULT,R1
    MOV X:(R0)+,D4.S Y:(R4)+,D7.S
    DO #NMUL,_ENDLOOP
    FMPY.S D4,D7,D0 X:(R0)+,D4.S Y:(R4)+,D7.S
    MOV D0.S,X:(R1)+
    _ENDLOOP
    ENDM
```

When a macro call is executed, the dummy arguments within the macro definition (NMUL, AVEC, BVEC, RESULT in the example above) are replaced with the corresponding argument as defined by the macro call.

All local labels within a macro are considered distinct for the currently active level of macro expansion (unless the macro local label override is used, see below). These local labels are valid for the entire macro expansion and are not considered bounded by non-local labels. Therefore, all local labels within a macro must be unique. This mechanism allows the programmer to freely use local labels within a macro definition without regard to the number of times that the macro is expanded. Non-local labels within a macro expansion are considered to be normal labels and thus cannot occur more than once unless used with the SET directive (see Chapter 6).
When specifying a local label within the body of a macro, the programmer must be aware that the label symbol is valid for the entire body of the current level of macro expansion. It is not valid for any nested macros within the current level of expansion. The example above shows why the local label feature is useful. If the macro N_R_MUL were called several times, there would be several _ENDLOOP labels resulting from the macro expansions. This is acceptable because each _ENDLOOP label is considered private to a particular instance of macro expansion.

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

5.4 Macro Calls

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

```
[label>     <macro name> [arguments] [comment]
```

The argument field can have the form:

```
[arg],[arg],...,<arg>]
```

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

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

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

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

5.5 Dummy Argument Operators

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

5.5.1 Dummy Argument Concatenation Operator - \n
Dummy arguments that are intended to be concatenated with other characters must be preceded by the concatenation operator, \ to separate them from the rest of the characters. The argument may precede or follow the adjoining text, but there must be no intervening blanks between the concatenation operator and the rest of the characters. To position an argument between two alphanumeric characters, place a backslash both before and after the argument name. For example, consider the following macro definition:

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

If this macro were called with the following statement,

```
SWAP_REG 0,1
```

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

```
MOVE R0,X0
MOVE R1,R0
MOVE X0,R1
```
5.5.2 Return Value Operator - ?

Another macro definition operator is the question mark (?) that returns the value of a symbol. When the macro processor encounters this operator, the ?<symbol> sequence is converted to a character string representing the decimal value of the <symbol>. For example, consider the following modification of the SWAP_REG macro described above:

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

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

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

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

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

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

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

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

```
MOVE R0,X0
MOVE R1,R0
MOVE X0,R1
```
5.5.3 Return Hex Value operator - %

The percent sign (%) is similar to the standard return value operator except that it returns the hexadecimal value of a symbol. When the macro processor encounters this operator, the %<symbol> sequence is converted to a character string representing the hexadecimal value of the <symbol>. Consider the following macro definition:

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

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

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

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

```
HEXA NOP
```

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

5.5.4 Dummy Argument String Operator - "

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

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

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

```
STR_MAC ABCD
```

then the resulting macro expansion would be:

```
DC 'ABCD'
```
Double quotes also make possible DEFINE directive expansion within quoted strings. Because of this overloading of the double quotes, care must be taken to insure against inappropriate expansions in macro definitions. Since DEFINE expansion occurs before macro substitution, any DEFINE symbols are replaced first within a macro dummy argument string:

```
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'
```

5.5.5 Macro Local Label Override Operator - ^

It may be desirable to pass a local label as a macro argument to be used as an address reference within the macro body. If a circumflex (^) precedes an expression containing an underscore label, during macro expansion the associated term will be evaluated with normal local label scope rather than macro call scope. Such interpretation disables the usual local label semantics for this particular reference within the macro call. Here is an example:

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

The macro local label override operator causes the ADDR argument to be interpreted as a local label outside the macro if the expanded argument has a leading underscore. If there is no leading underscore on the actual argument then the override operator has no effect. Consider the following macro call:

```
_LOCAL
LOAD _LOCAL
```

Without the local label override in the macro definition, an error would occur at the macro call because a symbol _LOCAL was not defined in the body of the macro. Because the circumflex was used the value of _LOCAL gets moved to R0. Note that any arbitrary string may be used as the actual parameter to the LOAD macro. The override operator has an effect only with underscore labels. Care must be exercised, however, in not defining a
macro label called _LOCAL and attempting to reference it as in the above example. In that case the macro local label override operator prevents the assembler from seeing the local label definition for that reference, and an error would result.

5.6 **DUP, DUPA, DUPC, DUPF Directives**

The **DUP**, **DUPA**, **DUPC**, and **DUPF** directives are specialized macro forms. They can be thought of as a simultaneous definition and call of an unnamed macro. The source statements between the **DUP**, **DUPA**, **DUPC**, and **DUPF** directives and the **ENDM** directive follow the same rules as macro definitions, including (in the case of **DUPA**, **DUPC**, and **DUPF**) the dummy operator characters described previously. For a detailed description of these directives, refer to Chapter 6.

5.7 **Conditional Assembly**

Conditional assembly facilitates the writing of comprehensive source programs that can cover many conditions. Assembly conditions may be specified through the use of arguments in the case of macros, and through definition of symbols via the **DEFINE**, **SET**, and **EQU** directives. Variations of parameters can then cause assembly of only those parts necessary for the given conditions. The built-in functions of the assembler provide a versatile means of testing many conditions of the assembly environment (see Section 3.8 for more information on the assembler built-in functions).

Conditional directives can also be used within a macro definition to ensure at expansion time that arguments fall within a range of allowable values. In this way macros become self-checking and can generate error messages to any desired level of detail.

The conditional assembly directive **IF** has the following form:

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

A section 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>** had 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.
Chapter 6 Assembler Significant Characters and Directives

6.1 Introduction

This chapter describes the directives that are recognized by the Freescale 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
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
- `.ENDIF` - 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. 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. Structured programming directives are discussed separately in Chapter 7.
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 semicolons (;;;) 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

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

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 \textbf{DEFINE} symbols will be expanded within the string. In all other respects usage of double quotes is equivalent to that of single quotes.

\textbf{EXAMPLE:}

Consider the source fragment below:

\begin{verbatim}
DEFINE LONG 'short'
STR_MAC MACRO STRING
  MSG 'This is a LONG STRING'
  MSG "This is a LONG STRING"
ENDM
\end{verbatim}

If this macro were invoked as follows,

\begin{verbatim}
STR_MAC sentence
\end{verbatim}

then the resulting expansion would be:

\begin{verbatim}
MSG 'This is a LONG STRING'
MSG 'This is a short sentence'
\end{verbatim}
@  

**Function Delimiter**

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

**EXAMPLE:**

```assembly
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 ['DSP56300',3,5] ; ID = '56300'
```
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:

```assembly
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:

```assembly
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:

```assembly
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:

```assembly
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 Modulo or Reverse-carry. If the runtime location counter is not zero, this directive first advances the runtime location counter to a base address that is a multiple of $2^k$, where $2^k \geq <expression>$. 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 Modulo buffer is specified, the expression must fall within the range $2 \leq <expression> \leq m$, where $m$ is the maximum address of the target DSP. If a Reverse-carry buffer is designated and <expression> is not a power of two a warning 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   $2FF-@LCV(R),$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 $2 \leq <expression> \leq m$, 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,$FFFFFFF ; INITIALIZE BUFFER TO ALL ONES
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 **Modulo** or **Reverse-carry**. If the runtime location counter is not zero, this directive first advances the runtime location counter to a base address that is a multiple of $2^k$, where $2^k \geq <expression>$. 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 **Modulo** buffer is specified, the expression must fall within the range $2 \leq <expression> \leq m$, where $m$ is the maximum address of the target DSP. If a **Reverse-carry** buffer is designated and `<expression>` is not a power of two a warning 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 D).

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 ARRAYSIZE '10 * SAMPLSIZE'
```

then the source line below:

```
DS ARRAYSIZE
```

would be transformed by the assembler to the following:

```
DS 10 * SAMPLSIZE
```
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 $2^k \geq <expression>$. 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,

<table>
<thead>
<tr>
<th>COUNT</th>
<th>SET</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>DUP</td>
<td>COUNT</td>
<td>; ASR BY COUNT</td>
</tr>
<tr>
<td>ASR</td>
<td>D0</td>
<td></td>
</tr>
<tr>
<td>ENDM</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

would generate the following in the source listing:

<table>
<thead>
<tr>
<th>COUNT</th>
<th>SET</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>DUP</td>
<td>COUNT</td>
<td>; ASR BY COUNT</td>
</tr>
<tr>
<td>ASR</td>
<td>D0</td>
<td></td>
</tr>
<tr>
<td>ASR</td>
<td>D0</td>
<td></td>
</tr>
<tr>
<td>ASR</td>
<td>D0</td>
<td></td>
</tr>
<tr>
<td>ENDM</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Note that the lines

\[
\text{DUP COUNT ;ASR BY COUNT ENDM}
\]

will only be shown on the source listing if the \textbf{MD} option is enabled. The lines

\[
\text{ASR D0 ASR D0 ASR D0}
\]

will only be shown on the source listing if the \textbf{MEX} option is enabled.

See the \textbf{OPT} directive in this chapter for more information on the \textbf{MD} and \textbf{MEX} options.
DUPA
Duplicate Sequence With Arguments

[label>] DU PA <dummy>,<arg>[<,<arg>,...,<arg>]
.
.
EN DM

The block of source statements defined by the DU PA and EN DM 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 DU PA directive processing.

See also: DUP, DUPC, DUPF, EN DM, MACRO

EXAMPLE:

If the input source file contained the following statements,

DU PA VALUE,12,32,34
DC VALUE
EN DM

then the assembled source listing would show

DU PA VALUE,12,32,34
DC 12
DC 32
DC 34
EN DM
Chapter 6 Assembler Significant Characters and Directives - Assembler Directives

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 \(<\text{increment}>\) is 1. \(<\text{start}>\) is the starting value for the loop index; \(<\text{end}>\) represents the final value. \(<\text{increment}>\) is the increment for the loop index; it defaults to 1 if omitted (as does the \(<\text{start}>\) value). The \(<\text{dummy}>\) parameter holds the loop index value and may be used within the body of instructions.

\(<\text{label}>\), 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,

\begin{verbatim}
DUPF NUM,0,7
MOVE #0,R\NUM
ENDM
\end{verbatim}

then the assembled source listing would show:

\begin{verbatim}
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
\end{verbatim}
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.
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 Program or None. 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
```
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
```
ENDEF

End of Conditional Assembly

ENDEF

The ENDEF 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 ENDEF 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

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
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

EQU [{X: | Y: | L: | P:}]<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 None, 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 None 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 COM-PUTE.
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
  ENDF
  .
  .
  .
ENDM
**FAIL**

**Programmer Generated Error**

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

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). <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).
Chapter 6 Assembler Significant Characters and Directives - Assembler Directives

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
List the Assembly

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
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). <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
Chapter 6 Assembler Significant Characters and Directives - Assembler Directives

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 <ABSOLUTE | RELATIVE>

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
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
Chapter 6 Assembler Significant Characters and Directives - Assembler Directives

Reporting Options

These options control what is reported in the listing file:

- CEX  - Print DC expansions
- CL   - Print conditional assembly directives
- 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
- MSW  - Warn on memory space incompatibilities
- UR   - Flag unresolved references
- W    - Display warning messages

Symbol Options

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

- DEX  - Expand DEFINE symbols within quoted strings
- 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
- SO   - Write symbols to object file
- XLL  - Write local labels to object file
- XR   - Recognize XDEFed symbols without XREF
Assembler Operation

Miscellaneous options having to do with internal assembler operation:

- **CC** - Enable cycle counts
- **CK** - Enable checksumming
- **CM** - Preserve comment lines within macros
- **CONST** - Make EQU symbols assembly time constants
- **CONTCK** - Continue checksumming
- **DLD** - Do not restrict directives in loops
- **GL** - Make all section symbols global
- **GS** - Make all sections global static
- **INTR** - Perform interrupt location checks
- **LB** - Byte increment load counter
- **LDB** - Listing file debug
- **MI** - Scan MACLIB directories for include files
- **PS** - Pack strings
- **PSM** - Programmable short addressing mode
- **RP** - Generate NOP to accommodate pipeline delay
- **RSV** - Check reserve data memory locations
- **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.
- **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. This option, if used, must be specified before the first symbol in the source program is defined.

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.

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.

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.
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.

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.

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

NOAE  Do not check address expressions.

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.
**Assembled Characters and Directives**

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

**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.

**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.

**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.

**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.

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

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

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

**NOSI** (DSP56300 only) (default, reset) Interpret an eight-bit short immediate value moved to a fractional register as a short unless forced long.
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.

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.

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

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 (DSP56300 only) Interpret an eight-bit short immediate value moved to a fractional register as a long unless forced short.

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.
Chapter 6 Assembler Significant Characters and Directives - Assembler Directives

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 XDEFed symbols to be recognized within other sections without being XREFed. 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
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, or P) 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.

<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, or P) 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.

\(<\text{lcc}>\)

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

\(<\text{imp}>\)

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.

\(<\text{ice}>\)

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.

\(<\text{exp2}>\)

Initial value to assign to the load counter used as the \(<\text{lcc}>\). If \(<\text{exp2}>\) is a relative expression the assembler uses the relative location counter. If \(<\text{exp2}>\) is an absolute expression the assembler uses the absolute location counter. If \(<\text{exp2}>\) 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 \texttt{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.
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:

\[ \text{BLANK\_TOP} + \text{BLANK\_BOTTOM} \leq \text{PAGE\_LENGTH} - 10 \]
\[ \text{BLANK\_LEFT} < \text{PAGE\_WIDTH} \]

See also: \textbf{LSTCOL}

**EXAMPLE:**

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

Purge Macro Definition

PNAME <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:

PNAME 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 relo-
cation. 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 di-
rective 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 sec-
tion needs data hiding, but independent relocation is not required.

If the LOCAL qualifier follows the <section name> in the SECTION directive, then all sym-
bs 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**

<text>

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'
```
SYM OBJ

Write Symbol Information to Object File

SYM OBJ  \(<\text{symbol}>[,\text{<symbol>},...,\text{<symbol>}]\)

The SYM OBJ directive causes information for each \(<\text{symbol}>\) to be written to the object file. This directive is recognized but currently performs no operation in COFF assemblers (see Appendix D, ).

A label is not allowed with this directive.

EXAMPLE:

SYM OBJ  XSTART,HIRTN,ERRPROC
TABS
Set Listing Tab Stops

TABS \(<\text{tabstops}>\)

The \text{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: \text{LSTCOL}

EXAMPLE:

\begin{verbatim}
TABS 4 ; Set listing file tab stops to 4
\end{verbatim}
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
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
Chapter 7 Structured Control Statements

7.1 Introduction

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

7.2 Structured Control Directives

The following directives are used for structured control. Note the leading period, which distinguishes these keywords from other directives and mnemonics. Structured control directives may be specified in either upper or lower case, but they must appear in the opcode field of the instruction line (e.g. they must be preceded either by a label, a space, or a tab).

.BREAK  .ENDI  .LOOP
.CONTINUE .ENDL  .REPEAT
.ELSE    .ENDW  .UNTIL
.ENDF    .FOR   .WHILE
          .IF

In addition, the following keywords are used in structured control statements:

AND  DOWNTO  TO
BY  OR
DO  THEN

Note that AND, DO, and OR are reserved assembler instruction mnemonics.
Chapter 7 Structured Control Statements - Syntax

7.3 Syntax

The formats for the .BREAK, .CONTINUE, .FOR, .IF, .LOOP, .REPEAT, and .WHILE statements are given in sections 7.3.4 through 7.3.2. Syntactic variables used in the formats are defined as follows:

- `<expression>` A simple or compound expression (section 7.4).
- `<stmtlist>` Zero or more assembler directives, structured control statements, or executable instructions.

Note that an assembler directive (Chapter 6) occurring within a structured control statement is examined exactly once -- at assembly time. Thus the presence of a directive within a .FOR, .LOOP, .REPEAT, or .WHILE statement does not imply repeated occurrence of an assembler directive; nor does the presence of a directive within an .IF-THEN-.ELSE structured control statement imply conditional assembly.

- `<op1>` A user-defined operand whose register/memory location holds the .FOR loop counter. The effective address must use a memory alterable addressing mode (e.g. it cannot be an immediate value).
- `<op2>` The initial value of the .FOR loop counter. The effective address may be any mode, and may represent an arbitrary assembler expression (Chapter 3).
- `<op3>` The terminating value of the .FOR loop counter. The effective address may be any mode, and may represent an arbitrary assembler expression (Chapter 3).
- `<op4>` The step (increment/decrement) of the .FOR loop counter each time through the loop. If not specified, it defaults to a value of #1. The effective address may be any mode, and may represent an arbitrary assembler expression (Chapter 3).
- `<cnt>` The terminating value in a .LOOP statement. This can be any arbitrary assembler expression (Chapter 3).

All structured control statements may be followed by normal assembler comments on the same logical line.

7.3.1 .BREAK Statement

SYNTAX:

.BREAK
FUNCTION:

The `.BREAK` statement causes an immediate exit from the innermost enclosing loop construct (.WHILE, .REPEAT, .FOR, .LOOP).

NOTES:

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

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

EXAMPLE:

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

### 7.3.2 `.CONTINUE` Statement

SYNTAX:

`.CONTINUE`

FUNCTION:

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

NOTES:

If a `.CONTINUE` is encountered with no loop statement active, a warning is issued.
.CONTINUE should be used with care near .ENDL directives or near the end of DO loops. It generates a jump instruction which is illegal in those contexts.

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

EXAMPLE:

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

7.3.3 .FOR Statement

SYNTAX:

.FOR <op1> = <op2> {TO | DOWNTO} <op3> [BY <op4>] [DO]
<stmtlist>
.ENDF

FUNCTION:

Initialize <op1> to <op2> and perform <stmtlist> until <op1> is greater (TO) or less than (DOWNTO) <op3>. Makes use of a user-defined operand, <op1>, to serve as a loop counter. .FOR-TO allows counting upward, while .FOR-DOWNTO allows counting downward. The programmer may specify an increment/decrement step size in <op4>, or elect the default step size of #1 by omitting the BY clause. A .FOR-TO loop is not executed if <op2> is greater than <op3> upon entry to the loop. Similarly, a .FOR-DOWNTO loop is not executed if <op2> is less than <op3>.

NOTES:

<op1> must be a writable register or memory location. It is initialized at the beginning of the loop, and updated at each pass through the loop. Any immediate operands must be preceded by a pound sign (#). Memory refer-
ences must be preceded by a memory space qualifier (X:, Y:, or P:). Local memory references are not allowed. Operands must be or refer to single-word values.

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

The DO keyword is optional.

EXAMPLE:

```
.FOR X:CNT = #0 TO Y:(targ*2)+114 ; loop on X:CNT
.
.
.
.ENDF
```

7.3.4 .IF Statement

SYNTAX:

```
.IF <expression> [THEN]
<stmtlist>
[ELSE
<stmtlist>]
.ENDI
```

FUNCTION:

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

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

EXAMPLE:

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

7.3.5 .LOOP Statement

SYNTAX:

```
.LOOP  <cnt>  
    <stmtlist>
.ENDL
```

FUNCTION:

Execute <stmtlist> <cnt> times. This is similar to the .FOR loop construct, except that the initial counter and step value are implied to be #1. It is actually a shorthand method for setting up a hardware DO loop on the DSP, without having to worry about addressing modes or label placement.

NOTES:

Since the .LOOP statement generates instructions for a hardware DO loop, the same restrictions apply as to the use of certain instructions near the end of the loop, nesting restrictions, etc.

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

EXAMPLE:

```
.LOOP  LPCNT           ; hardware loop LPCNT times
   ...
   ...
   ...
.ENDL
```
7.3.6  .REPEAT Statement

SYNTAX:

```
.REPEAT
<stmtlist>
.UNTIL <expression>
```

FUNCTION:

<stmtlist> is executed repeatedly until <expression> is true. When expression becomes true, advance to the next instruction following .UNTIL.

NOTES:

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

EXAMPLE:

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

7.3.7  .WHILE Statement

SYNTAX:

```
.WHILE <expression> [DO]
<stmtlist>
.ENDW
```

FUNCTION:

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

NOTES:

If <expression> is false upon entry to the .WHILE loop, <stmtlist> is not executed; execution continues after the .ENDW directive.
Chapter 7 Structured Control Statements - Simple and Compound Expressions

The DO keyword is optional.

EXAMPLE:

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

7.4 Simple and Compound Expressions

Expressions are an integral part of .IF, .REPEAT, and .WHILE statements. Structured control statement expressions should not be confused with the assembler expressions discussed in Chapter 3. The latter are evaluated at assembly time and will be referred to here as "assembler expressions"; they can serve as operands in structured control statement expressions. The structured control statement expressions described below are evaluated at run time and will be referred to in the following discussion simply as "expressions".

A structured control statement expression may be simple or compound. A compound expression consists of two or more simple expressions joined by either AND or OR (but not both in a single compound expression).

7.4.1 Simple Expressions

Simple expressions are concerned with the bits of the Condition Code Register (CCR). These expressions are of two types. The first type merely tests conditions currently specified by the contents of the CCR (section 7.4.1.1). The second type sets up a comparison of two operands to set the condition codes, and afterwards tests the codes (section 7.4.1.2).

7.4.1.1 Condition Code Expressions

A variety of tests (identical to those in the Jcc instruction) may be performed, based on the CCR condition codes. The condition codes, in this case, are preset by either a user-generated instruction or a structured operand-comparison expression (section 7.4.1.2). Each test is expressed in the structured control statement by a mnemonic enclosed in angle brackets; the mnemonics are described in Appendix E, Condition Code Mnemonics.
When processed by the assembler, the expression generates an inverse conditional jump to beyond the matching `ENDx/.UNTIL` directive. For example:

```assembly
.IF <EQ>;zero bit set?
+     bne Z_L00002 ;code generated by assembler
CLR D1 ;user code
.ENDI
+     Z_L00002 ;assembler-generated label
.REPEAT ;subtract until D0 < D7
+     Z_L00034 ;assembler-generated label
SUB D7,D0 ;user code
.UNTIL <LT>
+     bge Z_L00034 ;code generated by assembler
```

### 7.4.1.2 Operand Comparison Expressions

Two operands may be compared in a simple expression, with subsequent transfer of control based on that comparison. Such a comparison takes the form:

```
<op1>  <cc>  <op2>
```

where `<cc>` is a condition mnemonic enclosed in angle brackets (as described in section 7.4.1.1), and `<op1>` and `<op2>` are register or memory references, symbols, or assembler expressions. When processed by the assembler, the operands are arranged such that a compare/jump sequence of the following form always results:

```
CMP <reg1>,<reg2>
(JIB)cc <label>
```

where the jump conditional is the inverse of `<cc>`. Ordinarily `<op1>` is moved to the `<reg1>` data register and `<op2>` is moved to the `<reg2>` data register prior to the compare. This is not always the case, however: if `<op1>` happens to be `<reg2>` and `<op2>` is `<reg1>`, an intermediate register is used as a scratch register. In any event, worst case code generation for a given operand comparison expression is generally two moves, a compare, and a conditional jump.

Jumps or branches generated by structured control statements are forced long because the number and address of intervening instructions between a control statement and its termination are not known by the assembler. The programmer may circumvent this behavior by use of the `SCSJMP` directive (see Chapter 6).

Any immediate operands must be preceded by a pound sign (`#`). Memory references must be preceded by a memory space qualifier (X:, Y:, or P:). L memory references are not allowed. Operands must be or refer to single-word values.
Chapter 7 Structured Control Statements - Statement Formatting

Note that values in the <reg1> and <reg2> data registers are not saved before expression evaluation. This means that any user data in those registers will be overwritten each time the expression is evaluated at runtime. The programmer should take care either to save needed contents of the registers, reassign data registers using the SCSREG directive, or not use them at all in the body of the particular structured construct being executed. The data registers used by the structured control statements are listed in Appendix E.

7.4.2 Compound Expressions

A compound expression consists of two or more simple expressions (section 7.4.1) joined by a logical operator (AND or OR). The boolean value of the compound expression is determined by the boolean values of the simple expressions and the nature of the logical operator. Note that the result of mixing logical operators in a compound expression is undefined:

```
.IF X1 <GT> B AND <LS> AND R1 <NE> R2 ;this is OK
.IF X1 <LE> B AND <LC> OR R5 <GT> R6 ;undefined
```

The simple expressions are evaluated left to right. Note that this means the result of one simple expression could have an impact on the result of subsequent simple expressions, because of the condition code settings stemming from the assembler-generated compare.

If the compound expression is an AND expression and one of the simple expressions is found to be false, any further simple expressions are not evaluated. Likewise, if the compound expression is an OR expression and one of the simple expressions is found to be true, any further simple expressions are not evaluated. In these cases, the compound expression is either false or true, respectively, and the condition codes reflect the result of the last simple expression evaluated.

7.5 Statement Formatting

The format of structured control statements differs somewhat from normal assembler usage. Whereas a standard assembler line is split into fields separated by blanks or tabs, with no white space inside the fields, structured control statement formats vary depending on the statement being analyzed. In general, all structured control directives are placed in the opcode field (with an optional label in the label field) and white space separates all distinct fields in the statement. Any structured control statement may be followed by a comment on the same logical line.

7.5.1 Expression Formatting

Given an expression of the form:

```
<op1> <LT> <op2> OR <op3> <GE> <op4>
```
there must be white space (blank, tab) between all operands and their associated operators, including boolean operators in compound expressions. Moreover, there must be white space between the structured control directive and the expression, and between the expression and any optional directive modifier (THEN, DO). An assembler expression (Chapter 3) used as an operand in a structured control statement expression must not have white space in it, since it is parsed by the standard assembler evaluation routines:

```
.IF  #@CVI(@SQT(4.0)) <GT> #2 ; no white space in first operand
```

### 7.5.2 .FOR/.LOOP Formatting

The .FOR and .LOOP directives represent special cases. The .FOR structured control statement consists of several fields:

```
.FOR  <op1>  =  <op2>  TO  <op3>  BY  <op4>  DO
```

There must be white space between all operands and other syntactic entities such as =, TO, BY, and DO. As with expression formatting, an assembler expression used as an operand must not have white space in it:

```
.FOR  X:CNT  =  #0  TO  Y:(targ*2)+1  BY  #@CVI(@POW(2.0,@CVF(R)))
```

In the example above, the .FOR loop operands represented as assembler expressions (symbol, function) do not have embedded white space, whereas the loop operands are always separated from structured control statement keywords by white space.

The count field of a .LOOP statement must be separated from the .LOOP directive by white space. The count itself may be any arbitrary assembler expression, and therefore must not contain embedded blanks.

### 7.5.3 Assembly Listing Format

Structured control statements begin with the directive in the opcode field; any optional label is output in the label field. The rest of the statement is left as is in the operand field, except for any trailing comment; the X and Y data movement fields are ignored. Comments following the statement are output in the comment field (unless the unreported comment delimiter is used; see Chapter 6).

Statements are expanded using the macro facilities of the assembler. Thus the generated code can be sent to the listing by specifying the MEX assembler option, either via the OPT directive (Chapter 6) or the -O command line option (Chapter 1).

### 7.6 Effects on the Programmer’s Environment

During assembly, global labels beginning with "Z_L" are generated. They are stored in the symbol table and should not be duplicated in user-defined labels. Because these non-local labels ordinarily are not visible to the programmer there can be problems when local
(underscore) labels are interspersed among structured control statements. The **SCL** option (see the **OPT** directive, Chapter 6) causes the assembler to maintain the current local label scope when a structured control statement label is encountered.

In the `.FOR` loop, `<op1>` is a user-defined symbol. When exiting the loop, the memory/register assigned to this symbol contains the value which caused the exit from the loop.

A compare instruction is produced by the assembler whenever two operands are tested in a structured statement. At runtime, these assembler-generated instructions set the condition codes of the CCR (in the case of a loop, the condition codes are set repeatedly). Any user-written code either within or following a structured statement that references CCR directly (move) or indirectly (conditional jump/transfer) should be attentive to the effect of these instructions.

Jumps or branches generated by structured control statements are forced long because the number and address of intervening instructions between a control statement and its termination are not known by the assembler. The programmer may circumvent this behavior by use of the **SCSJMP** directive (see Chapter 6).

In all structured control statements except those using only a single condition code expression, registers are used to set up the required counters and comparands. In some cases, these registers are effectively reserved; the `.FOR` loop uses two data registers to hold the step and target values, respectively, and performs no save/restore operations on these registers. The assembler, in fact, does no save/restore processing in any structured control operation; it simply moves the operands into appropriate registers to execute the compare. See Appendix E for a list of registers used by the assembler in support of structured control statements on a particular processor. The **SCSREG** directive (Chapter 6) may be used to reassign structured control statement registers. The **MEX** assembler option (see the **OPT** directive, Chapter 6) may be used to send the assembler-generated code to the listing file for examination of possible register use conflicts.
## Appendix A ASCII Character Codes

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>ASCII</th>
<th>Decimal</th>
<th>Hex</th>
<th>ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>NUL</td>
<td>64</td>
<td>40</td>
<td>@</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>SOH</td>
<td>65</td>
<td>41</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>STX</td>
<td>66</td>
<td>42</td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td>03</td>
<td>ETX</td>
<td>67</td>
<td>43</td>
<td>C</td>
</tr>
<tr>
<td>4</td>
<td>04</td>
<td>EOT</td>
<td>68</td>
<td>44</td>
<td>D</td>
</tr>
<tr>
<td>5</td>
<td>05</td>
<td>ENQ</td>
<td>69</td>
<td>45</td>
<td>E</td>
</tr>
<tr>
<td>6</td>
<td>06</td>
<td>ACK</td>
<td>70</td>
<td>46</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>07</td>
<td>BEL</td>
<td>71</td>
<td>47</td>
<td>G</td>
</tr>
<tr>
<td>8</td>
<td>08</td>
<td>BS</td>
<td>72</td>
<td>48</td>
<td>H</td>
</tr>
<tr>
<td>9</td>
<td>09</td>
<td>HT</td>
<td>73</td>
<td>49</td>
<td>I</td>
</tr>
<tr>
<td>10</td>
<td>0A</td>
<td>LF</td>
<td>74</td>
<td>4A</td>
<td>J</td>
</tr>
<tr>
<td>11</td>
<td>0B</td>
<td>VT</td>
<td>75</td>
<td>4B</td>
<td>K</td>
</tr>
<tr>
<td>12</td>
<td>0C</td>
<td>FF</td>
<td>76</td>
<td>4C</td>
<td>L</td>
</tr>
<tr>
<td>13</td>
<td>0D</td>
<td>CR</td>
<td>77</td>
<td>4D</td>
<td>M</td>
</tr>
<tr>
<td>14</td>
<td>0E</td>
<td>S0</td>
<td>78</td>
<td>4E</td>
<td>N</td>
</tr>
<tr>
<td>15</td>
<td>0F</td>
<td>S1</td>
<td>79</td>
<td>4F</td>
<td>O</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>DLW</td>
<td>80</td>
<td>50</td>
<td>P</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>DC1</td>
<td>81</td>
<td>51</td>
<td>Q</td>
</tr>
<tr>
<td>18</td>
<td>12</td>
<td>DC2</td>
<td>82</td>
<td>52</td>
<td>R</td>
</tr>
<tr>
<td>19</td>
<td>13</td>
<td>DC3</td>
<td>83</td>
<td>53</td>
<td>S</td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>DC4</td>
<td>84</td>
<td>54</td>
<td>T</td>
</tr>
<tr>
<td>21</td>
<td>15</td>
<td>NAK</td>
<td>85</td>
<td>55</td>
<td>U</td>
</tr>
<tr>
<td>22</td>
<td>16</td>
<td>SYN</td>
<td>86</td>
<td>56</td>
<td>V</td>
</tr>
<tr>
<td>23</td>
<td>17</td>
<td>ETB</td>
<td>87</td>
<td>57</td>
<td>W</td>
</tr>
<tr>
<td>24</td>
<td>18</td>
<td>CAN</td>
<td>88</td>
<td>58</td>
<td>X</td>
</tr>
<tr>
<td>25</td>
<td>19</td>
<td>EM</td>
<td>89</td>
<td>59</td>
<td>Y</td>
</tr>
<tr>
<td>26</td>
<td>1A</td>
<td>SUB</td>
<td>90</td>
<td>5A</td>
<td>Z</td>
</tr>
<tr>
<td>27</td>
<td>1B</td>
<td>ESC</td>
<td>91</td>
<td>5B</td>
<td>[</td>
</tr>
<tr>
<td>28</td>
<td>1C</td>
<td>FS</td>
<td>92</td>
<td>5C</td>
<td>\</td>
</tr>
<tr>
<td>29</td>
<td>1D</td>
<td>GS</td>
<td>93</td>
<td>5D</td>
<td>]</td>
</tr>
<tr>
<td>30</td>
<td>1E</td>
<td>RS</td>
<td>94</td>
<td>5E</td>
<td>^</td>
</tr>
<tr>
<td>31</td>
<td>1F</td>
<td>US</td>
<td>95</td>
<td>5F</td>
<td>_</td>
</tr>
<tr>
<td>32</td>
<td>20</td>
<td>SP</td>
<td>96</td>
<td>60</td>
<td>`</td>
</tr>
<tr>
<td>33</td>
<td>21</td>
<td>!</td>
<td>97</td>
<td>61</td>
<td>a</td>
</tr>
<tr>
<td>34</td>
<td>22</td>
<td>&quot;</td>
<td>98</td>
<td>62</td>
<td>b</td>
</tr>
<tr>
<td>35</td>
<td>23</td>
<td>#</td>
<td>99</td>
<td>63</td>
<td>c</td>
</tr>
</tbody>
</table>
### Appendix A ASCII Character Codes

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>ASCII</th>
<th>Decimal</th>
<th>Hex</th>
<th>ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
<td>24</td>
<td>$</td>
<td>100</td>
<td>64</td>
<td>d</td>
</tr>
<tr>
<td>37</td>
<td>25</td>
<td>%</td>
<td>101</td>
<td>65</td>
<td>e</td>
</tr>
<tr>
<td>38</td>
<td>26</td>
<td>&amp;</td>
<td>102</td>
<td>66</td>
<td>f</td>
</tr>
<tr>
<td>39</td>
<td>27</td>
<td>'</td>
<td>103</td>
<td>67</td>
<td>g</td>
</tr>
<tr>
<td>40</td>
<td>28</td>
<td>(</td>
<td>104</td>
<td>68</td>
<td>h</td>
</tr>
<tr>
<td>41</td>
<td>29</td>
<td>)</td>
<td>105</td>
<td>69</td>
<td>i</td>
</tr>
<tr>
<td>42</td>
<td>2A</td>
<td>*</td>
<td>106</td>
<td>6A</td>
<td>j</td>
</tr>
<tr>
<td>43</td>
<td>2B</td>
<td>+</td>
<td>107</td>
<td>6B</td>
<td>k</td>
</tr>
<tr>
<td>44</td>
<td>2C</td>
<td>,</td>
<td>108</td>
<td>6C</td>
<td>l</td>
</tr>
<tr>
<td>45</td>
<td>2D</td>
<td>-</td>
<td>109</td>
<td>6D</td>
<td>m</td>
</tr>
<tr>
<td>46</td>
<td>2E</td>
<td>.</td>
<td>110</td>
<td>6E</td>
<td>n</td>
</tr>
<tr>
<td>47</td>
<td>2F</td>
<td>/</td>
<td>111</td>
<td>6F</td>
<td>o</td>
</tr>
<tr>
<td>48</td>
<td>30</td>
<td>0</td>
<td>112</td>
<td>70</td>
<td>p</td>
</tr>
<tr>
<td>49</td>
<td>31</td>
<td>1</td>
<td>113</td>
<td>71</td>
<td>q</td>
</tr>
<tr>
<td>50</td>
<td>32</td>
<td>2</td>
<td>114</td>
<td>72</td>
<td>r</td>
</tr>
<tr>
<td>51</td>
<td>33</td>
<td>3</td>
<td>115</td>
<td>73</td>
<td>s</td>
</tr>
<tr>
<td>52</td>
<td>34</td>
<td>4</td>
<td>116</td>
<td>74</td>
<td>t</td>
</tr>
<tr>
<td>53</td>
<td>35</td>
<td>5</td>
<td>117</td>
<td>75</td>
<td>u</td>
</tr>
<tr>
<td>54</td>
<td>36</td>
<td>6</td>
<td>118</td>
<td>76</td>
<td>v</td>
</tr>
<tr>
<td>55</td>
<td>37</td>
<td>7</td>
<td>119</td>
<td>77</td>
<td>w</td>
</tr>
<tr>
<td>56</td>
<td>38</td>
<td>8</td>
<td>120</td>
<td>78</td>
<td>x</td>
</tr>
<tr>
<td>57</td>
<td>39</td>
<td>9</td>
<td>121</td>
<td>79</td>
<td>y</td>
</tr>
<tr>
<td>58</td>
<td>3A</td>
<td>:</td>
<td>122</td>
<td>7A</td>
<td>z</td>
</tr>
<tr>
<td>59</td>
<td>3B</td>
<td>;</td>
<td>123</td>
<td>7B</td>
<td>{</td>
</tr>
<tr>
<td>60</td>
<td>3C</td>
<td>&lt;</td>
<td>124</td>
<td>7C</td>
<td>l</td>
</tr>
<tr>
<td>61</td>
<td>3D</td>
<td>=</td>
<td>125</td>
<td>7D</td>
<td>}</td>
</tr>
<tr>
<td>62</td>
<td>3E</td>
<td>&gt;</td>
<td>126</td>
<td>7E</td>
<td>~</td>
</tr>
<tr>
<td>63</td>
<td>3F</td>
<td>?</td>
<td>127</td>
<td>7F</td>
<td>DEL</td>
</tr>
</tbody>
</table>
Appendix B Directive Summary

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

B.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
- RDIrections - 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
B.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

B.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

B.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
- **PRCTRL**: Send control string to printer
- **STITLE**: Initialize program subtitle
- **TABS**: Set listing tab stops
- **TITLE**: Initialize program title
Appendix B Directive Summary - Object File Control

B.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

B.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

B.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
Appendix C Assembler Messages

C.1 Introduction

Assembler messages are grouped into four categories:

Command Line Errors

These errors indicate invalid command line options, missing filenames, file open errors, or other invocation errors. Command line errors generally cause the assembler to stop processing.

Warnings

Warnings notify the programmer of suspect constructs but do not otherwise affect the object file output.

Errors

These errors indicate problems with syntax, addressing modes, or usage. In these cases the resulting object code is generally not valid.

Fatal

Fatal errors signify serious problems encountered during the assembly process such as lack of memory, file not found, or other internal errors. The assembler halts immediately.

The assembler also will provide information on the source field location of the error, if it can be ascertained. If a listing file is produced, messages ordinarily will appear immediately before the line containing the error. One exception is when the relationship between the first and last instructions in a DO loop produces an error. In this case the error text will appear after the last instruction at the end of the loop. Messages are always routed to standard output.
Appendix C Assembler Messages - Command Line Errors

C.2 Command Line Errors

Cannot open command file
Cannot open listing file
Cannot open object file

The file associated with a -F, -L, or -B command line option was not found.

Cannot open source file

The assembly source input file was not found.

Duplicate listing file specified - ignored
Duplicate object file specified - ignored

The -L and -B command line options were encountered more than once on the command line. Only the first occurrence of each option is used. The assembler continues processing.

Illegal command line -D option argument

The symbol name given in a -D command line option is invalid (possibly too long or does not begin with an alphabetic character), or the substitution string is not enclosed in single quotes.

Illegal command line -I option argument

A problem occurred when attempting to save the include file path string.

Illegal command line -M option argument

A problem occurred when attempting to save the MACLIB file path string.

Illegal command line -P option argument

The string provided as the processor type is not valid.

Illegal command line -R option argument

The string provided as the revision level is not valid.

Illegal command line option

The option specified on the command line was not recognized by the assembler.

Interrupted

The assembler was interrupted by a keyboard break (Control-C).
LDB option with no listing file specified; using source file

The **LDB** option was given on the command line without a corresponding `-L` to generate a listing file. If a listing file is not produced, debugging packages cannot use it for source tracking. Therefore the assembler uses the default assembly language file as input for source tracking.

**Missing command line option argument**

The expected arguments following a command line option specifier were missing.

**Missing source filename**

There must be at least one source filename specified on the command line.

**Source file name same as listing file name**

**Source file name same as object file name**

One of the source files appeared to the assembler to have the same name as the specified listing or object file. The assembler aborts rather than potentially writing over a source input file.
Appendix C Assembler Messages - Warnings

C.3 Warnings

Absolute address involves incompatible memory spaces

The memory space attribute is regarded by the assembler as a type, in the same sense that high level languages use type for variables. Symbols may have memory space attributes of X, Y, L, P(ogram), or N(one); only N is fully compatible with all other attributes. In this case, an operand was evaluated with a different memory space attribute than that specified in the instruction.

Absolute address too large to use I/O short - long substituted

The absolute address is not within the range specifying an I/O short address, even though the I/O short forcing operator has been used. The assembler substitutes long absolute addressing.

Absolute address too large to use short - long substituted

The absolute address value being forced short will not fit in the storage allocated for a short address. The assembler substitutes long absolute addressing.

Absolute address too small to use I/O short - long substituted

The absolute address is not within the range specifying an I/O short address, even though the I/O short forcing operator has been used. The assembler substitutes long absolute addressing.

Address involves incompatible memory spaces

The memory space attribute is regarded by the assembler as a type, in the same sense that high level languages use type for variables. Symbols may have memory space attributes of X, Y, L, P(ogram), or N(one); only N is fully compatible with all other attributes. In this case, an operand was evaluated with a different memory space attribute than that specified in the instruction.

Cannot force short addressing for source and destination

In a MOVEP or MOVES instruction an attempt was made to force both operands to short or I/O short. The second operand defaults to long.

Cannot force short immediate with this parallel move

The immediate operand in the X field of a parallel X memory and register move cannot be forced short. The mode is changed to long immediate.

Contents of assigned register in previous instruction not available - generating NOP instruction

Due to pipelining, if an address register (Rn or Nn) is changed in the previous instruction, the new contents are not available for use as a pointer until the next instruction. If the RP option is in effect (see the OPT directive, Chapter 6) the
assembler produces this warning and generates a **NOP** prior to the offending instruction.

**Debug directives ignored - use command line debug option**

A source-level debug directive was encountered but the assembler command line -G option was not given.

**Destination operand assumed I/O short**

Neither operand in a **MOVEP** instruction is explicitly declared I/O short; however, the source operand does not qualify, so the destination operand is assumed to be the I/O short operand.

**Directive not allowed in command line absolute mode**

The **MODE** directive is ignored when the assembler command line -A option is active.

**Duplicate listing file specified - ignored**
**Duplicate object file specified - ignored**

The -B or -L command line options were given more than once.

**ENDDO instruction not inside DO loop**

An **ENDDO** instruction was found outside the scope of an active **DO** loop.

**Explicit bottom margin ignored with page length of zero**
**Explicit top margin ignored with page length of zero**

The top or bottom margin parameters to a **PAGE** directive are ignored because the page length parameter was zero.

**Expression involves incompatible memory spaces**

The memory space attribute is regarded by the assembler as a type, in the same sense that high level languages use type for variables. Symbols may have memory space attributes of **X**, **Y**, **L**, **P**rogram, or **N**(one); only **N** is fully compatible with all other attributes. In this case, two operands were evaluated with different memory space attributes, neither of which was **N**.

**Expression value outside fractional domain**

The expected fractional value was not within the range \(-1.0 \leq m < 1\).  

**Extra dimensions ignored**

A source-level debug **.DIM** directive contained more than four arguments specifying the dimensions of an array.
Appendix C Assembler Messages - Warnings

FMPY suffix ignored - using secondary opcode suffix

The single or extended precision suffix on a dual-opcode instruction has been ignored. The precision is taken from the suffix on the secondary opcode.

Immediate value too large to use short - long substituted

An immediate data value being forced short is too large to fit in the space allocated for a short immediate value. The assembler substitutes long immediate addressing.

Improper nesting of DO loops

The end address of a subordinate (nested) DO loop is greater than or equal to the end address of the loop enclosing it. The end address of a nested DO instruction must be less than the end address of the enclosing loop.

Instruction cannot appear in interrupt vector locations

Some instructions cannot be used reliably as interrupt code. These instructions include RTI, RTS, DO, and ENDDO.

Instruction does not allow data movement specified - using MOVE encoding
Instruction does not allow data movement specified - using MOVEP encoding
Instruction does not allow data movement specified - using MOVES encoding

An inappropriate MOVE-type instruction was written given the type of the operands. The assembler substitutes a valid encoding for the operands in question.

Invalid destination register for this instruction - using TFR3 encoding

The destination register for a TFR2 instruction was not either X or Y.

Invalid interrupt vector address - using SWI address

Currently only the SWI address is supported by the TRAPcc instruction.

I/O short absolute address cannot be forced

I/O short absolute addressing is not valid for this operation. An appropriate addressing mode (long absolute, short jump, short absolute) is substituted.

I/O short absolute address cannot be forced - long substituted

I/O short absolute addressing is not valid for this operation. The assembler substitutes long absolute addressing.

I/O short address cannot be forced

I/O short addressing is not valid for this operation. An appropriate addressing mode (long, short, short jump) is substituted.
I/O short address cannot be forced - long substituted

I/O short addressing is not valid for this operation. The assembler substitutes long addressing.

Label field ignored

The assembler directive does not allow a label, so the assembler will not store the label value in the symbol table.

Load location counter overflow
Load location counter underflow

The load location counter exceeded its maximum or minimum value. The assembler wraps the counter value around and continues.

Load origin involves incompatible memory spaces

The memory space attribute is regarded by the assembler as a type, in the same sense that high level languages use type for variables. Symbols may have memory space attributes of X, Y, L, P rog(ram), or N(one); only N is fully compatible with all other attributes. In this case, an operand was evaluated with a different memory space attribute than that specified in the directive.

Load reserved address space violation

The load location counter has incremented into a reserved area of data address space.

Long absolute address cannot be forced

Long absolute addressing is not valid for this operation. An appropriate addressing mode (I/O absolute, short jump, short absolute) is substituted.

Long absolute address cannot be forced - substituting I/O short addressing

Long absolute addressing is not valid for this operation. The assembler substitutes I/O short addressing.

Long absolute address cannot be forced - substituting short addressing

Long absolute addressing is not valid for this operation. The assembler substitutes short absolute addressing.

Long immediate cannot be forced

Long immediate data is not valid for this operation. An appropriate size for the target DSP is substituted.
Appendix C Assembler Messages - Warnings

Long PC-relative address cannot be forced
Long PC-relative addressing is not valid for this operation. An appropriate addressing mode (short PC-relative) is substituted.

Macro expansion not active
A macro must have been called prior to using the @ARG() or @CNT() functions (see Chapter 3).

Macro name is the same as existing assembler directive
Macro name is the same as existing assembler mnemonic
The name of the macro being defined conflicts with the name of an assembler directive or mnemonic. Either use a different macro name or use the RDIRECT directive to remove the directive or mnemonic name from the assembler lookup tables.

No control registers accessed - using MOVE encoding
A MOVEC-type instruction was given but no control registers were used as operands. The assembler substitutes a valid encoding for the operands in question.

No looping construct found - .BREAK ignored
No looping construct found - .CONTINUE ignored
A .BREAK or .CONTINUE structure control statement was encountered outside of any active looping construct (.FOR, .LOOP, .REPEAT, .WHILE).

Number of macro expansion arguments is greater than definition
Number of macro expansion arguments is less than definition
A discrepancy exists between the number of arguments specified in a macro definition and the number of arguments provided in the macro call.

Options for both debug and strip specified - strip ignored
Both the -G and -Z options were given on the command line. The -G option takes precedence.

P space not accessed - using MOVE encoding
A MOVEP-type instruction was given but a P memory reference was not used as an operand. The assembler substitutes a valid encoding for the operands in question.

Page directive with no arguments ignored with page length of zero
A PAGE directive with no arguments, which ordinarily produces a form feed in the listing output, is ignored because a previous PAGE directive specified a page length of zero.
Appendix C Assembler Messages - Warnings

PC-relative address involves incompatible memory spaces

The memory space attribute is regarded by the assembler as a type, in the same sense that high level languages use type for variables. Symbols may have memory space attributes of X, Y, L, P (rogram), or N (one); only N is fully compatible with all other attributes. In this case, an operand was evaluated with a different memory space attribute than that specified in the instruction.

PC-relative address too large to use short - long substituted

The PC-relative offset is not within the range specifying a short PC-relative offset, even though the short forcing operator has been used. The assembler substitutes long PC-relative addressing.

Post-update operation will not occur on destination register

If the source operand in a MOVE operation specifies a post-update addressing mode and the destination register is the same as the source operand register then the post-update operation will not take place.

PRCTL directive ignored - no explicit listing file

The PRCTL directive takes effect only if the -L option is used on the command line to explicitly specify a listing file.

Redefinition of symbol

A symbol used in a DEFINE directive has been redefined without an intervening UNDEF directive. The assembler discards the previous definition and replaces it with the new definition.

Rounding not available with LMS move - using MAC/MPY encoding

A MACR or MPYR instruction was specified in conjunction with LMS move syntax.

Runtime location counter overflow

Runtime location counter underflow

The runtime location counter exceeded its maximum or minimum value. The assembler wraps the counter value around and continues.

Runtime origin involves incompatible memory spaces

The memory space attribute is regarded by the assembler as a type, in the same sense that high level languages use type for variables. Symbols may have memory space attributes of X, Y, L, P (rogram), or N (one); only N is fully compatible with all other attributes. In this case, an operand was evaluated with a different memory space attribute than that specified in the directive.
Appendix C Assembler Messages - Warnings

Runtime reserved address space violation
The runtime location counter has incremented into a reserved area of data address space.

Short absolute address cannot be forced
Short absolute addressing is not valid for this operation. An appropriate addressing mode (long absolute) is substituted.

Short absolute address cannot be forced - long substituted
Short absolute addressing is not valid for this operation. The assembler substitutes long absolute addressing.

Short immediate cannot be forced
Short immediate data is not valid for this operation. An appropriate size is substituted.

Short PC-relative address cannot be forced
Short PC-relative addressing is not valid for this operation. An appropriate addressing mode (long PC-relative) is substituted.

Signed operand must come first in signed/unsigned combinations
In a MPYSU, MACSU, or DMACSU instruction, the signed operand must come first in the operand ordering.

Source operand assumed I/O short
Neither operand in a MOVEP instruction is explicitly declared I/O short; however, the destination operand does not qualify, so the source operand is assumed to be the I/O short operand.

Storage block size not a power of 2
The expression in a DSR directive did not evaluate to a power of 2. Since the DSR directive is generally used to allocate FFT buffers for reverse carry operations, the size of the buffer may be in error.

String truncated in expression evaluation
Only the first four characters of a string constant are used during expression evaluation except for arguments to the DC directive (see Chapter 6).

Strip not valid in relocatable mode - ignored
The -Z option is valid only when the -A option is also given.
Appendix C Assembler Messages - Warnings

Unresolved external reference

Ordinarily the assembler does not flag unresolved references in relative mode, assuming they will be resolved at link time. If the UR option is specified, the assembler will generate this warning if any symbols are undefined during the second pass.
Appendix C Assembler Messages - Errors

C.4 Errors

Absolute address contains forward reference - force short or I/O short address

The assembler attempted a default to short addressing which failed. Either the absolute address is too large or it needs to be forced I/O short.

Absolute address must be either short or I/O short

The absolute address is too large for a short address and outside the range of valid I/O short addresses.

Absolute address too large to use I/O short
Absolute address too small to use I/O short

The absolute address being forced short is outside the range of valid I/O short addresses. This usually means that the I/O short address has not been ones extended.

Absolute address too large to use short

The absolute address value is too large to be forced short.

Absolute addressing mode not allowed

Absolute operands are not allowed with some instructions, in particular parallel XY data memory moves.

Address mode syntax error - expected ‘)’
Address mode syntax error - expected ‘+’
Address mode syntax error - expected ‘+’ or ‘-’
Address mode syntax error - expected comma
Address mode syntax error - expected comma or end of field
Address mode syntax error - expected offset register
Address mode syntax error - extra characters
Address mode syntax error - probably missing ‘)’

A syntax error was detected when scanning the source line operand and/or X and Y data fields. These errors may indicate omission of a source operand, insufficient white space between fields, or improper specification of address register indirect addressing modes.

Argument outside function domain

An argument to one of the transcendental built-in functions was inappropriate.

Arithmetic exception

An internal floating point exception occurred while evaluating an expression. The result of the evaluation is probably not valid.
Assembler directive or mnemonic not found

An argument to the RDIRECT directive was not a recognized assembler directive or mnemonic.

Base argument larger than machine word size

The base parameter of a @FLD() function has a value larger than can fit in the target machine word.

Binary constant expected

A character other than ASCII '0' or '1' either followed the binary constant delimiter (%) or appeared in an expression where a binary value was expected by default.

Bit mask cannot span more than eight bits

If the first operand of a BFxxx-type instruction was shifted one bit to the right until the low-order bit was a 1, the resulting value must not exceed $FF hexadecimal.

Cannot conditionally repeat write to memory

A move to memory cannot be preceded by a REPcc-type instruction.

Cannot nest section inside itself

A section of a given name may not have another SECTION directive with the same name declared inside it.

Cannot nest symbol definitions

A source-level debug .DEF directive was encountered inside another .DEF-.END-DEF pair.

Cannot open include file

The specified INCLUDE file cannot be found, or the operating system limit on open files has been exceeded.

Cannot repeat this instruction

Cannot repeat two-word instruction

The REP instruction cannot be used to repeat two-word instructions or instructions that change program flow. Instructions that cannot be repeated include DO, Jcc, JCLR, JMP, JSET, JScc, JSCLR, JSR, JSSET, REP, RTI, RTS, and SWI.

CONST option must be used before any label

This option, which prevents EQU symbols from being exported to the object file, must be given before any label is encountered in the source file.
Appendix C Assembler Messages - Errors

Contents of assigned register in previous instruction not available

Due to pipelining, if an address register (Rn or Nn) is changed in the previous instruction, the new contents are not available for use as a pointer until the next instruction.

Count must be an integer value

The argument to a DUP directive did not evaluate as an integer expression.

CRE option must be used before any label

The CRE option must be activated before any labels are encountered so that the assembler can append cross-reference data to all applicable symbol table entries.

Data allocation exceeds buffer size

Data allocated between a BUFFER-ENDBUF sequence exceeded the size specified in the BUFFER directive.

Decimal constant expected

A character other than ASCII '0' through '9' either followed the decimal constant delimiter ('.') or appeared in an expression where a decimal value was expected by default.

DEFINE symbol must be a global symbol name

A local label (a symbol beginning with the underscore character) may not be used as a DEFINE directive symbol.

Displacement address mode not allowed

Long displacement addressing is not allowed with some instructions, in particular parallel XY data memory moves.

Divide by zero

The expression evaluator detected a divide by zero.

DO loop address must be in current section

The loop address of a DO loop cannot fall outside the bounds of its enclosing section. This is particularly important in relative mode as the loop address is calculated based on the starting address of the section.

Dummy argument not found

The dummy argument name given as an argument to the @ARG() function was not found in the macro dummy argument list.
Duplicate destination register not allowed

If the opcode-operand portion of an instruction specifies a destination register, the same register or portion of that register may not be specified as a destination in the parallel data bus move operation.

Duplicate source and destination register not allowed

If the opcode-operand portion of an instruction specifies a source register, the same register or portion of that register may not be specified as a destination in the parallel data bus move operation.

Either source or destination memory space must be X or Y

One of the operands in a \textbf{MOVEP} instruction must reference a location in X or Y memory.

ELSE without associated IF directive

An \textbf{ELSE} directive was encountered without a preceding \textbf{IF} conditional assembly directive.

.ELSE without associated .IF directive

An \textbf{.ELSE} directive was encountered before a matching \textbf{.IF} conditional structured control statement.

Empty bit mask field

The first operand of a \textbf{BFxxx}-type instruction was zero.

End of structure or union without matching definition

A source-level debug end-of-structure symbol declaration was recognized without a preceding structure or union definition.

ENDBUF without associated BUFFER directive

An \textbf{ENDBUF} directive was encountered without a preceding \textbf{BUFFER} directive.

.ENDEF without associated .DEF directive

A source-level debug \textbf{.ENDEF} directive was encountered without a preceding \textbf{.DEF} directive.

.ENDIF without associated .IF directive

An end-of-conditional or end-of-loop directive was encountered before a matching conditional or loop structured control statement.
ENDIF without associated IF directive

An **ENDIF** directive was encountered without a preceding **IF** conditional assembly directive.

.ENDL without associated .LOOP directive

An end-of-loop directive was encountered before a matching loop structured control statement.

ENDM without associated MACRO directive

An **ENDM** directive was encountered without a preceding **MACRO** directive.

ENDSEC without associated SECTION directive

An **ENDSEC** directive was encountered without a preceding **SECTION** directive.

.ENDW without associated .WHILE directive

An end-of-loop directive was encountered before a matching loop structured control statement.

EQU requires label

The **EQU** directive must have a label to associate with the equated expression.

EXITM without associated MACRO directive

An **EXITM** directive was encountered without a preceding **MACRO** directive.

Expression cannot have a negative value

Some directives do not allow negative expression arguments, as for example in the **PAGE** directive controls.

Expression contains forward references

Some directives do not allow expression arguments which have not yet been defined in the source, as for example in the **IF**, **EQU**, or **SET** directives.

Expression involves incompatible memory spaces

The memory space attribute is regarded by the assembler as a type, in the same sense that high level languages use type for variables. Symbols may have memory space attributes of **X**, **Y**, **L**, **P**(rogram), or **N**(one); only **N** is fully compatible with all other attributes. In this case, two operands were evaluated with different memory space attributes, neither of which was **N**.

Expression must be greater than zero

Some directives require a nonzero argument, as for example in the **BSC** directive.
Expression result must be absolute

Certain directives and some assembler usage require absolute values as arguments or operands.

Expression result must be integer

Certain directives and some assembler usage require integer values as arguments or operands.

Expression result too large

The expression evaluated to a value greater than the acceptable range. This error can occur when an expression result exceeds the native word size of the target DSP.

External reference not allowed in expression

References to external symbols (e.g. symbols not defined in the current assembly source input) are not allowed in some types of byte or integer expressions.

External reference not allowed in function

References to external symbols (e.g. symbols not defined in the current assembly source input) are not allowed as direct or indirect arguments to any built-in function.

Extra characters beyond expression

The expression evaluator found extra characters after the end of a valid expression. Unbalanced parentheses can cause this error.

Extra characters following string

An end-of-string delimiter was followed by unexpected characters on the source line.

Extra characters following symbol name

A non-alphanumeric character other than the underscore (_) was encountered in a symbol name.

Extra characters in function argument or missing ')' for function

Mismatched parentheses or wrong number of parameters in a function invocation.

Extra characters in operand field

The PAGE directive contains too many operands.

Extra fields ignored

There were extra fields specified in an assembler directive.
First data move destination accumulator same as operand destination accumulator
The destination of the data move field is the same as the Data ALU destination.

First data move field required with this instruction
A TFR3 instruction requires a register and a data move operand field.

First data move source accumulator same as operand destination accumulator
The source of the data move field is the same as the Data ALU destination.

Floating point constant expected
A character other than ASCII '0' through '9', 'e' or 'E', or '.' appeared in an expression where a floating point value was expected by default.

Floating point not allowed in relative expression
Relative expressions are generally used for address computation, therefore a floating point value would not be appropriate.

Floating point value not allowed
An immediate value expressed in floating point notation is only valid in a MOVE-type instruction.

Forcing not specified
The type of forcing operand was not given in a FORCE directive.

Function result out of range
The result computed by a transcendental function was too large to be represented on the host machine.

GL option must be used before any section
The GL option must be activated before any explicit sections are encountered so that the assembler can insure that all section symbols are global.

GLOBAL without preceding SECTION directive
A GLOBAL directive was encountered outside any previously defined section.

GS option must be used before any section
The GS option must be activated before any explicit sections are encountered so that the assembler can use the appropriate counters for section relocation.
Appendix C Assembler Messages - Errors

**Hex constant expected**

A character other than ASCII '0' through '9', 'a' through 'f', or 'A' through 'F' either followed the hexadecimal constant delimiter ($) or appeared in an expression where a hexadecimal value was expected by default.

**IC option must be used before any symbol, section, or macro definition**

The IC option must be activated before any symbols, sections, or macros are defined so that the assembler can remain consistent when storing label names in the symbol table.

**IDENT directive must contain revision number**

**IDENT directive must contain version number**

The version and revision numbers are both required arguments for the IDENT directive.

**Illegal directive in buffer declaration**

A directive was encountered between a BUFFER-ENDBUF pair that is not allowed in that context. Some invalid directives include any other buffer-type directive (DSM, DSR, etc.), section directives, or any directive which alters the current location counter designation (MODE, ORG).

**Illegal directive inside .DEF-.ENDEF declaration**

**Illegal directive outside .DEF-.ENDEF declaration**

Some source-level debug directives, such as .FILE, make no sense and are not allowed inside .DEF-.ENDEF declarations. Conversely, other directives such as .VAL are not allowed outside of a .DEF-.ENDEF declaration.

**Illegal directive inside DO loop**

A directive was encountered inside a DO loop that is not allowed in that context. Some invalid directives include any buffer-type directive (DSM, DSR, etc.), section directives, or any directive which alters the current location counter designation (MODE, ORG).

**Illegal function argument**

An invalid argument was passed to one of the assembler built-in functions, in particular the @LCV() function.

**Illegal instruction in single-instruction DO loop**

A conditional break instruction (BRKcc) cannot be used as the only instruction in a DO loop.
Illegal memory counter specified
The memory counter designation supplied in the ORG directive was not one of H (high), L (low), or a positive integer expression in parentheses.

Illegal memory map character
The memory map character supplied in the ORG directive was not one of I (internal), E (external), R (ROM), A (port A), or B (port B).

Illegal memory space specified
Illegal memory space specified - L:
Illegal memory space specified - P:
Illegal memory space specified - X:
Illegal memory space specified - Y:
The memory space given is either invalid or inappropriate for the desired operation.

Illegal move field destination specified
Illegal move field destination register specified
The destination operand in a data memory move is invalid for the type of instruction specified.

Illegal move field source specified
The source operand in a data memory move is invalid for the type of instruction specified.

Illegal operator for floating point element
Bitwise operators are invalid for floating point values.

Illegal option
An argument to the OPT directive is invalid.

Illegal processor type
The argument to the -P command line option is invalid.

Illegal revision
The argument to the -R command line option is invalid.

Illegal secondary mnemonic
The secondary mnemonic to an FMPY instruction was not one of FADD, FSUB, or FADDSUB.
Illegal use of SSH as loop count operand

The contents of the system stack high register may not be used as the loop count operand of a **DO** instruction.

Illegal X field destination specified
Illegal X field destination register specified
Illegal Y field destination specified

The destination operand in an X or Y memory data move is invalid for the type of instruction specified.

Illegal Y field source specified

The source operand in an X memory data move is invalid for the type of instruction specified.

Immediate addressing mode not allowed

Immediate operands are not allowed with some instructions, in particular program memory moves (**MOVEM**).

Immediate operand not allowed

Immediate operands are not allowed with some instructions, in particular program memory moves (**MOVEM**).

Immediate operand required

The **ANDI** and **ORI** instructions must have an immediate value as the source operand.

Immediate value too large

The immediate operand value is too large for the space allotted in the instruction.

Immediate value too large to use short

The immediate value being forced short is too large to fit into the instruction word.

Increment value cannot be zero

The increment parameter to a **DUPF** directive must be greater than zero.

Indexed address mode not allowed

**XY** parallel data moves and the **LEA** instruction do not allow indexed addressing mode.
Initial debug directive must be .FILE

In a source file containing debug directives being assembled with the -G option the .FILE directive must be the first source-level debug directive in the input stream.

Instruction cannot appear at last address of a DO loop
Instruction cannot appear at next to last address of a DO loop
Instruction cannot appear within last 2 words of a DO loop
Instruction cannot appear within last 3 words of a DO loop

Some instructions are restricted within a variable range of the DO instruction loop address. These instructions include DO, ENDDO, JMP, Jcc, JCLR, JSET, and moves to or from particular control registers.

Instruction cannot appear immediately after control register access

Some instructions must not appear immediately after certain control registers have been accessed. These instructions include RTI, RTS, DO, and ENDDO.

Instruction does not allow data movement specified

The desired operation may only be done with a MOVE instruction.

Invalid address expression

An attempt was made to evaluate an expression consisting of two relative terms with the same sign.

Invalid addressing mode

The addressing mode of one of the operands in the instruction was not recognized.

Invalid buffer type

The buffer type specified in a BADDR or BUFFER directive was not one of M (modulo) or R (reverse-carry).

Invalid conditional register transfer syntax

The syntax for an IFcc or FFcc conditional address register move was incorrect.

Invalid destination register

The first data move destination register in a double memory read operation was not valid.

Invalid dummy argument name

Macro argument names cannot be local symbols, e.g. they cannot begin with the underscore (_) character.
Invalid force type

The argument to a **FORCE** directive must be SHORT, LONG, or NONE.

Invalid function name

The name following the function invocation character (@) was not recognized.

Invalid label field width specified

The argument given to the **LSTCOL** directive does not allow enough room on the listing line for the remaining fields to be output.

Invalid macro name

Macro names cannot be local symbols, e.g. they cannot begin with the underscore (_) character.

Invalid memory space attribute

The memory space attribute given is not one of the letters X, Y, L, or P.

Invalid mode

The mode specified in a **MODE** directive was not either RELATIVE or ABSOLUTE.

Invalid opcode field width specified
Invalid opcode 2 field width specified
Invalid operand field width specified
Invalid operand 2 field width specified

The argument given to the **LSTCOL** directive does not allow enough room on the listing line for the remaining fields to be output.

Invalid page length specified

The minimum page length allowed by the **PAGE** directive is 10 lines per page. The maximum is 255.

Invalid page width specified

The minimum page width allowed by the **PAGE** directive is 1 column per line. The maximum is 255.

Invalid radix expression

The expression in the **RADIX** directive does not evaluate to one of the supported constant bases (2, 8, 10, or 16).

Invalid register combination

The source operand registers in a **FMPY** instruction cannot be used together.
Appendix C Assembler Messages - Errors

Invalid register specified

The direct register operand is incorrect for this instruction.

Invalid relative expression

The terms of a relative expression may only participate in addition and subtraction operations and must have opposing signs.

Invalid secondary opcode

The opcode in the second operation field is not one of the instructions FADD, FSUB, or FADDSUB.

Invalid section directive modifier

The qualifier specified in a SECTION directive was not either GLOBAL or STATIC.

Invalid section name

Section names cannot be local symbols, e.g. they cannot begin with the underscore (_) character.

Invalid shift amount

A shift expression must evaluate to within the range 0 <= n <= m, where m is the maximum address of the target DSP.

Invalid source address mode

The source address mode in a MOVEP instruction was not valid.

Invalid source address register

Invalid source register

The source register in a double memory read operation was not valid.

Invalid storage class

The storage class given in a source-level debug symbol declaration is unknown.

Invalid tabs stops specified

The argument to the TAB directive is out of range.

Invalid X field destination address mode

Invalid X field source address mode

The address mode in the source or destination of the X data move field was invalid.
Invalid X field width specified
Invalid Y field width specified

The argument given to the LSTCOL directive does not allow enough room on the listing line for the remaining fields to be output.

Invalid XY address register specification

In some XY memory parallel data moves, if the register forming the effective address of the X data field is from the set R0-R3, the effective address register in the Y field must be from the set R4-R7. Conversely, if the register forming the effective address of the X data field is from the set R4-R7, the effective address register in the Y field must be from the set R0-R3.

Invalid XY data register specification

In some XY memory parallel data moves, if the data register of the X data field is from the set D0-D3, the data register in the Y field must be from the set D4-D7. Conversely, if the data register of the X data field is from the set D4-D7, the data register in the Y field must be from the set D0-D3.

I/O short addressing mode not allowed

An operand was forced I/O short when I/O short addressing was not allowed.

Jump based on SSH or SSL cannot follow update of SP
Jump via SSH or SSL cannot follow write to SP

A JSET, JCLR, JSSET, or JSCLR instruction which tested a bit in either the SSH or SSL register was immediately preceded by a MOVE to the SP register.

L space specified for load, but not for runtime
L space specified for runtime, but not for load

Since L memory space is the only double-wide memory space, if L memory space is the runtime memory space, the only valid load memory space is L. Likewise, L memory space can never be specified as the load memory space if runtime memory space is X, Y, or P.

LB option must be used before any code or data generation

The LB option must be specified before any code or data in order for the assembler to increment the location counter appropriately.

LDB option must be used before any code or data generation

The LDB option must be specified before any code or data in order for the assembler to establish the debug source file appropriately.
Left margin exceeds page width

The blank left margin value in the PAGE directive exceeds the default or specified page width parameter.

Length value greater than string size

The length parameter in a substring construct is larger than the composite length of the input string argument.

Line too long

Source statements, including continuation lines, cannot exceed 512 characters in length.

LOC option must be used before any local label

The LOC option must appear before any local label so that the assembler can keep the local label lists synchronized.

LOCAL directive not valid in global section

The LOCAL directive is not allowed inside an implicit or explicit global section.

Local symbol names cannot be used with GLOBAL
Local symbol names cannot be used with LOCAL
Local symbol names cannot be used with XDEF
Local symbol names cannot be used with XREF

Underscore labels are not allowed with this directive.

LOCAL without preceding SECTION directive

A LOCAL directive was encountered outside any previously defined section.

Long absolute address cannot be used

An operand was forced long where only a short or I/O short address was valid.

Long absolute cannot be used - force short or I/O short

A forward reference was forced long where only a short or I/O short address was valid.

Macro cannot be redefined

A macro name cannot be used as the label for a second macro definition in the same source file unless the macro is defined and used within a declared section (see the SECTION directive, Chapter 6).
Macro not defined
   The macro name was not found in the macro lookup table.

Macro value substitution failed
   The evaluation of a macro argument expression failed.

Memory bounds greater than maximum address
   The bounds argument in a LOMEM or HIMEM directive is invalid.

Memory counter designator value too large
   The integer counter designator in an ORG directive is greater than 65535.

Memory space must be P or NONE
   An END directive was encountered while the runtime memory space was X, Y, or L.

Missing ‘(’ for function
   All assembler built-in functions require at least one argument which must be enclosed in parentheses.

Missing ‘)’ in expression
   Parentheses are not balanced in an expression.

Missing argument
   The argument to a DUPA or DUPC directive was not found.

Missing definition string
   The substitution string for a DEFINE directive is missing.

Missing delimiter in substring
   A substring construct was missing the closing square bracket.

Missing dimension
   The .DIM directive had no arguments.

Missing directive name
   The argument to an RDIRECT directive is missing.

Missing expression
   An expression was expected by the expression evaluator.
Appendix C Assembler Messages - Errors

Missing filename
No filename was provided as an argument to the INCLUDE directive.

Missing line number
No line number was provided as an argument to the .LINE directive.

Missing macro name
A MACRO directive was encountered without a label or the macro name was omitted from a PMACRO directive.

Missing memory space specifier
One of the operands of an instruction was expected to have a memory space specifier (X:, Y:, L:, or P:) preceding the address mode specifier.

Missing option
The OPT directive was specified without an argument.

Missing or illegal memory space specifier
One of the operands of an instruction was expected to have a memory space specifier (X:, Y:, L:, or P:) preceding the address mode specifier.

Missing or mismatched quote
A single or double quote character was expected by the string parsing routines.

Missing pathname
No pathname was provided as an argument to the MACLIB directive.

Missing processor type
There was no argument provided for the -P command line option.

Missing quote
Missing quote in string
A single or double quote character was expected by the string parsing routines.

Missing revision
There was no argument provided for the -R command line option.

Missing section name
No section name was given as an argument to the SECTION directive.
Missing size argument
No size value was given as an argument to the .SIZE directive.

Missing string after concatenation operator
The string concatenation operator (++) must be followed by another quoted string.

Missing symbol name
The SYMOBJ, XDEF, and XREF directives require at least one symbol name as an argument.

Missing symbol value
No symbol value was given as an argument to the .VAL directive.

Missing tag for end of structure or union
A source-level debug structure or union declaration was found without a corresponding tag definition.

Missing tag name
No tag name was given as an argument to the .TAG directive.

Mnemonic must indicate precision using .S or .X suffix
Floating point instructions generally must indicate the precision of their operation by appending either .S for single precision or .X for single-extended precision.

Mode not specified
The MODE directive was not followed by either RELATIVE or ABSOLUTE.

Move from SSH or SSL cannot follow move to SP
Move from SSH or SSL cannot follow update to SP
A MOVE instruction using the system stack (SSH or SSL) as a source operand cannot immediately follow a MOVE which uses the stack pointer (SP) as a destination operand.

MU option must be used before any code or data generation
The MU option must be given before any data allocation directive (BSC, DC, DS, DSM, DSR) or any instruction appears in the source file.

Negative immediate value not allowed
The immediate count value for a DO or REP instruction cannot be less than zero.
Appendix C Assembler Messages - Errors

Negative or empty DO loop not allowed

The loop address given in a DO instruction must specify an address at least one greater than the current program counter value.

NOGS option must be used before any section

The NOGS option must be activated before any explicit sections are encountered so that the assembler can use the appropriate counters for section relocation.

No previous function declaration

A .EF debugging directive was encountered without a corresponding .BF directive.

Not enough fields specified for instruction

There were no operands specified for a MOVE, MOVEC, MOVEM, or MOVEP instruction.

No-update mode not allowed

The no-update register addressing mode is not allowed for this instruction (e.g., the LEA instruction).

Offset register number must be the same as address register number

The explicit offset register number in an operand using post-increment, post-decrement, or indexed by offset addressing mode (Nn) is different from the number specified for the address register (Rn). The offset register number may be omitted from these types of indirect addressing modes; the assembler defaults to the address register number.

Offset value greater than string size

The offset parameter in a substring construct is larger than the composite length of the input string argument.

Only absolute addressing allowed

The instruction allows only absolute addressing.

Only absolute and register direct addressing allowed

The instruction allows only absolute and register direct addressing.

Only immediate addressing allowed

The instruction allows an immediate source operand only.

Only immediate and register direct addressing allowed

The instruction allows only immediate and register direct addressing modes.
Appendix C Assembler Messages - Errors

Only immediate and register direct and indirect addressing allowed

The instruction allows only immediate, register direct, and register indirect addressing modes.

Only PC-relative addressing allowed

The instruction allows only PC-relative addressing.

Only PC-relative and register direct addressing allowed

The instruction allows only PC-relative and register direct addressing.

Only post-increment by offset addressing allowed with LMS move

The initial data move destination operand in an LMS move must use post-increment by offset addressing.

Only post-increment or post-increment by offset addressing allowed

Moves to P memory allow only post-increment or post-increment by offset addressing.

Only register direct addressing allowed

The instruction allows only register direct addressing.

Only register direct and indirect addressing allowed

The instruction allows only register direct and indirect addressing.

Only register indirect addressing allowed

The instruction allows only register indirect addressing.

Operation not allowed with address term

Only addition and subtraction are allowed in expressions involving addresses or relative terms.

Page length too small for specified top and bottom margins

The sum of the top and bottom margins specified in the PAGE directive is greater than the page length - 10.

Page length too small to allow default bottom margin

The bottom margin exceeds the page length specified in the PAGE directive.

PC-relative address too large to use short

The PC-relative offset being forced short is too large to fit into the instruction word.
Appendix C Assembler Messages - Errors

PC-relative addressing mode not allowed

The PC-relative addressing mode is not allowed for this instruction. The restriction applies, for example, to bit manipulation instructions and some jump-type instructions.

Phasing error

The value associated with a symbol has changed between pass 1 of the assembly and pass 2. This error can occur spontaneously in conjunction with other errors. The assembler is designed to avoid phasing errors in general. If a phasing error occurs without any other errors this may represent an internal error which should be reported to Freescale.

One exception is the use of the checksumming function @CHK() with the EQU directive. Instruction encoding may be incomplete after the first pass due to forward referencing, causing the checksum value to change between passes. Because of this the SET directive must be used to assign the checksum value to a symbol.

Possible invalid white space between operands or arguments

The assembler verifies that fields which should not contain operands or values are empty. If these fields are not empty the assembler produces this error.

Post-decrement addressing mode not allowed

The post-decrement addressing mode is not allowed for this instruction. The restriction applies, for example, to bit manipulation instructions and some jump-type instructions.

Post-decrement by offset addressing mode not allowed

The post-decrement by offset addressing mode is not allowed for this instruction. The restriction applies, for example, to bit manipulation instructions and some jump-type instructions.

Post-increment addressing mode not allowed

The post-increment addressing mode is not allowed for this instruction. The restriction applies, for example, to bit manipulation instructions and some jump-type instructions.

Post-increment by offset addressing mode not allowed

The post-increment by offset addressing mode is not allowed for this instruction. The restriction applies, for example, to bit manipulation instructions and some jump-type instructions.
Pre-decrement addressing mode not allowed

The pre-decrement addressing mode is not allowed for this instruction. The restriction applies, for example, to instructions which include parallel XY memory data transfers.

RDIRECT directive not allowed in section

Since the effect of the RDIRECT directive is global, it cannot be used within a section which has been declared using the SECTION directive. Move the RDIRECT directive outside the declared section to avoid this error.

Redefinition would overflow line

A substitution string declared using the DEFINE directive will cause the current source line to overflow if substitution occurs.

Reference outside of current buffer block
Reference outside of current overlay block

Reference was made to an underscore local label which fell outside the current buffer or overlay definition.

Register direct addressing not allowed

Register direct addressing mode is not allowed for this instruction. The restriction applies, for example, to bit manipulation instructions, some jump-type instructions, and parallel XY data memory moves.

Register displacement valid only with address register R2

Only address register R2 is valid as a displacement register.

Relative equate must be in same section

An EQU directive with a relative expression operand must be defined in the same section as the section associated with the operand expression.

Relative expression must be integer

A relative expression must evaluate to an integer value.

Relative expression not allowed

Relative expressions are not allowed as arguments to the assembler built-in functions.

Relative SET must be in same section

A SET directive with a relative expression operand must be defined in the same section as the section associated with the operand expression.
Relative terms from different sections not allowed

Relative terms defined in different sections are not allowed in expressions. This is because the relationship between the terms is based on where the enclosing sections are located in memory.

Reserved name used for symbol name

One of the DSP register names has been used as a label, operand, or directive argument. These register names, in either upper or lower case, are reserved by the assembler. See Appendix E for a list of DSP register names.

Runtime space must be P

An instruction was encountered and the runtime memory space was not set to P (Program).

SCO option must be used before any label

The SCO option sends structured control statements to the object file and thus must be specified before any symbols are defined in the source file.

Second data move destination accumulator same as operand destination accumulator

The destination accumulator in a double memory read instruction is the same as the DALU accumulator specification.

Secondary opcode not allowed

The secondary opcode field is allowed only with the FMPY instruction.

Section not encountered on pass 1

The section declared in a SECTION directive was not encountered during the first pass of the assembler. This situation indicates an internal assembler error and should be reported to Freescale.

SET requires label

The SET directive must have a label in order to associate the directive argument with a symbol name.

SET symbol names cannot be used with GLOBAL
SET symbol names cannot be used with LOCAL
SET symbol names cannot be used with XDEF

A symbol defined using the SET directive cannot be exported from a section using GLOBAL, LOCAL, or XDEF.
Appendix C Assembler Messages - Errors

Short absolute address too large

The flagged operand value is greater than the maximum short address of the target DSP.

Short I/O absolute address too large
Short I/O absolute address too small

The flagged operand value is outside the I/O address range of the target DSP.

Short or I/O short address expected

A short or I/O short address was expected as the second operand of a JCLR, JSET, JSCLR, or JSSET instruction.

Short PC-relative address too large

The flagged operand value is greater than the maximum PC-relative address of the target DSP.

SSH cannot be both source and destination register

In a MOVE instruction, the SSH register cannot be both the source and destination operand.

Start argument greater than machine word size

The start parameter of a @FLD() function has a value larger than can fit in the target machine word.

Start position greater than source string size

The start parameter in a @POS() function is larger than the total length of the source string argument.

Storage block size must be greater than zero

The size of a buffer allocated with the DSM, DSR, BSM, BSB, and other buffer directives was too small.

Storage block size out of range

The size of the buffer in a DSM, DSR, BSM, BSB, or other buffer directive is too large to be allocated.

Storage block too large

The runtime location counter overflowed while the assembler was attempting to allocate storage through a DSM or DSR directive. The assembler automatically advances the program counter to the next valid base address given the size of the modulo or reverse carry buffer. This error occurs when the sum of the expression
in the DSM or DSR directive and the runtime location counter value exceed available memory in the current memory space.

Structure or union tag mismatch

A matching tag name could not be found for the current source-level debug structure or union declaration.

Subroutine branch to loop address not allowed

A BSR, BScc, BSSET, or BSCLR instruction cannot have as its target the loop address of the current DO loop.

Subroutine jump to loop address not allowed

A JSR, JScc, JSSET, or JSCLR instruction cannot have as its target the loop address of the current DO loop.

SVO option must be used before any code or data generation

The SVO option must be given before any data allocation directive (BSC, DC, DS, DSM, DSR) or any instruction appears in the source file.

Symbol already defined as GLOBAL
Symbol already defined as LOCAL
Symbol already defined as XDEF
Symbol already defined as XREF

The symbol used in an GLOBAL, LOCAL, XDEF, or XREF directive has already been defined in a previous directive of the same type.

Symbol already defined as global

A symbol specified in an XDEF directive has already been defined as global outside the current section.

Symbol already defined in current section

A symbol specified in an XREF directive has already been defined as private within the current section.

Symbol already used as SET symbol

The label has already been used in a SET directive. A symbol defined with SET cannot be redefined except through another SET directive.

Symbol cannot be set to new value

The label has been defined previously other than with the SET directive. Only symbols defined using the SET directive may be redefined.
Symbol defined in current section before GLOBAL directive
Symbol defined in current section before XDEF directive

The GLOBAL or XDEF directive must appear within a section prior to the definition of any symbols in its argument list. Any symbols within a section which must be accessible outside the section should be declared in a GLOBAL or XDEF directive immediately following the SECTION directive.

Symbol name too long

Symbols are limited to 512 characters. The first character must be alphabetic or the underscore character (A-Z, a-z, _). The remaining characters must be alphanumeric, including the underscore character (A-Z, a-z, 0-9, _).

Symbol not previously defined

The symbol specified in an UNDEF directive was not previously defined in a DEFINE directive.

Symbol redefined

The symbol has already been used as a label in a previous context.

Symbol tag mismatch

A matching tag reference could not be found for a tagged symbol table entry.

Symbol undefined on pass 2

The symbol used as an operand or directive argument was never defined in the source program.

Symbols must start with alphabetic character

Symbol names must begin with an upper or lower case alphabetic character or the underscore character (_).

SYMOBJ symbol must be a global symbol name

Arguments to the SYMOBJ directive cannot be preceded by an underscore.

Syntax error - expected ‘)’:

In an ORG directive using numeric counter designations the parenthesis/colon pair separating the load or runtime address from the memory space, counter, or mapping characters was not found.

Syntax error - expected ‘:’

In an ORG directive the colon separating the load or runtime address from the memory space, counter, or mapping characters was not found.
Syntax error - expected '>

The closing angle bracket in a non-local INCLUDE directive argument was not found.

Syntax error - expected comma

The comma separating operands in an instruction or directive was not found.

Syntax error - expected keyword BY

In a .FOR structured control statement something other than the optional step clause preceded by the keyword BY was encountered.

Syntax error - expected keyword DO

In a .WHILE structured control statement something other than the optional DO keyword was encountered at the end of the statement.

Syntax error - expected keyword TO or DOWNTO

In a .FOR structured control statement something other than the loop target clause preceded by the keyword TO or DOWNTO was encountered.

Syntax error - expected quote

The assembler was expecting the start of a quoted string.

Syntax error - extra characters

Extra characters were found after an instruction or directive operand.

Syntax error - invalid assignment operator

The loop assignment operator in a .FOR structured control statement is not an equals sign (=).

Syntax error - invalid compound operator

Structured control statement compound operators are either AND or OR.

Syntax error - invalid conditional operator

The conditional operator in a structure control statement expression is not valid.

Syntax error - invalid statement terminator

There were extra or invalid characters found at the end of a structured control statement.

Syntax error - missing address mode specifier

An instruction operand was not specified.
Syntax error - missing operand

An operand in a structured control statement expression was missing.

Syntax error in directive name list

A character other than a comma was found separating the arguments in an RDI-RECT or SYMOBJ directive name list.

Syntax error in dummy argument list

A character other than a comma was found separating the dummy arguments in a macro definition (MACRO directive), or a dummy argument began with the underscore character (_).

Syntax error in macro argument list

A character other than a comma was found separating the arguments in a macro call.

Syntax error in macro name list

A character other than a comma was found separating the arguments in a PMACRO directive name list.

Syntax error in symbol name list

A character other than a comma was found separating the arguments in an XDEF or XREF directive name list.

Tag name not found

A matching tag name could not be found for the current source-level debug structure or union declaration.

Too many fields specified for instruction

An instruction field that was expected to be empty contained data other than a comment. This can happen when an instruction using only the X data transfer field encounters data other than a comment in the Y data transfer field.

Two dummy arguments are the same

Two dummy arguments in a macro definition (MACRO directive) have the same name.

UNDEF symbol must be a global symbol name

The argument to an UNDEF directive cannot be a local label, e.g. a name starting with the underscore character (_).
Appendix C Assembler Messages - Errors

Unexpected end of file - missing .ENDF
Unexpected end of file - missing .ENDI
Unexpected end of file - missing .ENDL
Unexpected end of file - missing .ENDW
Unexpected end of file - missing .UNTIL

The matching end-of-conditional or end-of-loop directive for a conditional or looping structured control statement was never found.

Unexpected end of file - missing COMMENT delimiter

The second occurrence of the delimiter character in a COMMENT directive was never found.

Unexpected end of file - missing ENDBUF

A BUFFER directive was encountered without a closing ENDBUF directive.

Unexpected end of file - missing ENDIF

An IF directive was encountered without a closing ENDIF directive.

Unexpected end of file - missing ENDM

A macro definition was started using the MACRO directive, but the end of the source file was encountered before a closing ENDM directive was found.

Unexpected end of file - missing ENDSEC

A SECTION directive was found without a closing ENDSEC directive.

Unknown math error

A transcendental math function returned an error that could not be classified as out of range or outside the function domain.

Unrecognized mnemonic

A symbol in the assembler opcode field was not a defined macro, an instruction mnemonic, or a directive.

Unrecognized secondary mnemonic

A symbol in the assembler secondary opcode field was not one of the instructions FADD, FSUB, or FADDSUB.

.UNTIL without associated .REPEAT directive

An .UNTIL directive was encountered before a matching .REPEAT structured control statement.
Value argument larger than machine word size

The value parameter of a \texttt{@FLD()} function has a value larger than can fit in the target machine word.

Width argument greater than machine word size

The width parameter of a \texttt{@FLD()} function has a value larger than can fit in the target machine word.

XDEF without preceding SECTION directive
XREF without preceding SECTION directive

An XDEF or XREF directive was encountered outside any previously defined section.

XLL option must be used before any local label

The XLL option must be activated before any local labels are encountered so that the assembler can make the appropriate entries in the symbol table.

XR option must be used before any label

The XR option must be activated before any labels are encountered so that the assembler can make the appropriate entries in the symbol table.
Appendix C Assembler Messages - Fatal Errors

C.5 Fatal Errors

<mode> encoding failure

A bad address mode indicator or register number was passed to the assembler encoding routines. <mode> represents the register set or addressing mode in question. This is a serious internal error that should be reported to Freescale.

Absolute mode select failure

The mode indicator passed to the absolute addressing mode selection logic was not valid. This is a serious internal error that should be reported to Freescale.

Arithmetic exception

An internal floating point exception occurred while evaluating an expression. The assembler cannot continue.

Cannot encode instruction
Cannot encode branch instruction
Cannot encode jump instruction

The correspondence between the source opcode mnemonic and the internal opcode type has been corrupted. This is an internal error that should be reported to Freescale.

Cannot seek to start of line number entries
Cannot seek to start of object data
Cannot seek to start of object file
Cannot seek to start of relocation entries
Cannot seek to start of section headers
Cannot seek to start of string table
Cannot seek to start of symbol table

An I/O error occurred which prevented the assembler from positioning correctly in the output object file.

Cannot write file header to object file
Cannot write line number entries to object file
Cannot write optional header to object file
Cannot write relocation entries to object file
Cannot write section headers to object file
Cannot write string table to object file
Cannot write symbols to object file

An I/O error occurred which prevented the assembler from writing data to the output object file.
Appendix C Assembler Messages - Fatal Errors

Cannot write control string to listing file
Cannot write left margin to listing file
Cannot write new line to listing file
Cannot write new page to listing file
Cannot write page header to listing file
Cannot write string to listing file

An I/O error occurred which prevented the assembler from writing data to the output listing file.

Compare select error

The comparison indicator passed to the evaluator selection logic was not valid. This is a serious internal error that should be reported to Freescale.

Debug symbol type failure

The symbol type indicator passed to the debug selection logic was not valid. This is a serious internal error that should be reported to Freescale.

Directive select error

The directive indicator passed to the directive selection logic was not valid. This is a serious internal error that should be reported to Freescale.

DO stack out of sequence

The assembler maintains an internal stack representing DO loop nesting levels. The internal stack pointers have been corrupted.

Error in mnemonic table

The indicator passed to the instruction processing logic was not valid. This is a serious internal error that should be reported to Freescale.

Expression operator failure

Expression operator lookup has failed. This is a serious internal error that should be reported to Freescale.

Expression stack underflow

An attempt has been made to free an expression when there are none to be freed. This is an internal error that should be reported to Freescale.

Fatal segmentation or protection fault
Contact Freescale DSP Operation

A program error has caused the assembler to access an invalid host system address. This generally indicates a bug in the assembler software.
Appendix C Assembler Messages - Fatal Errors

File info out of sequence

File debug information is scrambled. This is a serious internal error that should be reported to Freescale.

File not encountered on pass 1

The file in the source input list was never processed by the assembler during pass 1. This is an internal error that should be reported to Freescale.

Immediate mode select error

The mode indicator passed to the immediate addressing mode selection logic was not valid. This is a serious internal error that should be reported to Freescale.

Input mode stack out of sequence

The stack for recording whether input is from a file or a macro expansion has been corrupted. This is an internal error that should be reported to Freescale.

Invalid DO loop range check

The value passed to the end-of-DO-loop verification logic is bad. This is an internal error that should be reported to Freescale.

Invalid instruction class

The saved MAC-type instruction class has been corrupted. This is an internal error that should be reported to Freescale.

Invalid tag storage class

The saved tag storage class has been corrupted. This is an internal error that should be reported to Freescale.

I/O error writing data word to object file

An I/O error occurred which prevented the assembler from writing data to the output object file.

Location bounds selection failure

The logic for selecting the appropriate bounds array based on the current memory space has returned a bad value. This is an internal error that should be reported to Freescale.

Option select error

The option indicator passed to the option selection logic (OPT directive) was not valid. This is a serious internal error that should be reported to Freescale.
Out of memory - assembly aborted

There is not enough internal memory to perform dynamic storage allocation. Since the assembler keeps all working information in memory, including the symbol table and macro definitions, there is the possibility that memory will be exhausted if many symbols or macros are defined in a single assembly run.

PC-relative mode select failure

The mode indicator passed to the PC-relative addressing mode selection logic was not valid. This is a serious internal error that should be reported to Freescale.

Register selection failure

The register number passed to the multiply mask selection logic was not valid. This is a serious internal error that should be reported to Freescale.

Section counter sequence failure

The ordering of location counter structures has been corrupted. This is an internal error that should be reported to Freescale.

Section stack mode error

The assembler expected to restore a nested section but found the section list empty. This is an internal error that should be reported to Freescale.

Too many lines in source file

An individual source file contained more than 2**31 lines of code.

Too many sections in module

There is a limit of 255 discrete sections in a given source file.

Unrecognized transformation mnemonic

The lookup of an FADD or FSUB secondary operand failed. This is an internal error that should be reported to Freescale.
Appendix D Freescale DSP Object File Format (COFF)

D.1 Introduction

The Freescale DSP assembler and linker produce a binary object file in a modified form of the AT&T Common Object File Format (COFF). COFF is a formal definition for the structure of machine code files. It originated with Unix System V but has sufficient flexibility and generality to be useful in non-hosted environments. In particular, COFF supports user-defined sections and contains extensive information for symbolic software testing and debugging.

Later sections describe the COFF implementation for the Freescale family of digital signal processors. The DSP COFF format has been altered to support multiple memory spaces and normalized to promote transportability of object files among host processors. See section D.4 for a list of differences between the Freescale DSP object file format and standard COFF. For a more general discussion of COFF the following reference may be useful:


D.2 Object File Structure

A DSP COFF object file consists of up to eight groups of object file information. Some of these groups are optional, depending on the type of object file generated, and others may have repeating occurrences. The basic object file components are:

- File header
- Optional header
- Section headers
- Section data
- Relocation information
- Line numbers
- Symbol table
- String table
The general layout of the object file is illustrated in Figure E-1.

![Diagram of COFF File Basic Structure]

Figure E-1  COFF File Basic Structure

The file header contains object file information such as timestamp, number of sections, pointer to the symbol table, and file status flags. Depending on how the object file was generated the optional header holds link or run time information. The optional header is followed by a list of section headers. Each section header contains pointers to section data, relocation information, and line number entries. After the section headers comes the raw data for all sections. If the object file is relocatable the raw data may be followed by a block of relocation entries for all sections. If the original source file was compiled or assembled with the \texttt{-G} debug option, the relocation information is followed by source line number address entries. The symbol table contains information on program symbols useful by both the linker and the debugger. The string table may contain very long symbolic names, comment text, or relocation expressions. Note that the last four groups (relocation info, line number entries, symbol table, and string table) may not appear if the linker \texttt{-S} option is used to strip symbols from the object file.
D.3 Object File Components

Following are detailed descriptions of each of the DSP COFF object file components. The descriptions include the purpose of the component, its structure in the object file, and meanings of individual fields within the component.

D.3.1 File Header

The file header is the first component in a COFF object file. It contains information about the object file itself and is used for negotiating other components within the file. There is only one file header per object file. Figure E-2 shows the structure of the COFF file header.

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Declaration</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-3</td>
<td>unsigned long</td>
<td>f_magic</td>
<td>Magic number</td>
</tr>
<tr>
<td>4-7</td>
<td>unsigned long</td>
<td>f_nscns</td>
<td>Number of sections</td>
</tr>
<tr>
<td>8-11</td>
<td>long int</td>
<td>f_timdat</td>
<td>Time and date when file was created</td>
</tr>
<tr>
<td>12-15</td>
<td>long int</td>
<td>f_symptr</td>
<td>File pointer to the start of the symbol table</td>
</tr>
<tr>
<td>16-19</td>
<td>long int</td>
<td>f_nsyms</td>
<td>Number of symbol table entries</td>
</tr>
<tr>
<td>20-23</td>
<td>unsigned long</td>
<td>f_opthdr</td>
<td>Number of bytes in the optional header</td>
</tr>
<tr>
<td>24-27</td>
<td>unsigned long</td>
<td>f_flags</td>
<td>Flags (see Figure E-3)</td>
</tr>
</tbody>
</table>

Figure E-2 File Header Format

The magic number is a special code indicating the target machine for the object file (DSP56300, etc.). The number of sections is useful for scanning the list of section headers. The date and time stamp is kept in binary form and may contain a host-dependent time value. The f_symptr field contains a file byte offset to the beginning of the symbol table. The number of symbol table entries provides an upper bound for looping through the symbol table and an indirect means for accessing the start of the string table. The size of the optional header allows for jumping to the start of the section header list.
The flags field is a set of bit flags which convey status information about the object file. It is used primarily by linkers, debuggers, and other loader software to determine whether the file is valid for a particular requested operation. The individual bit flags are shown in Figure E-3.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>F_RELFLG</td>
<td>000001</td>
<td>Relocation information stripped from file</td>
</tr>
<tr>
<td>F_EXEC</td>
<td>000002</td>
<td>File is executable (no unresolved external references)</td>
</tr>
<tr>
<td>F_LNNO</td>
<td>000004</td>
<td>Line numbers stripped from file</td>
</tr>
<tr>
<td>F_LSYMS</td>
<td>000010</td>
<td>Local symbols stripped from file</td>
</tr>
<tr>
<td>F_CC</td>
<td>001000</td>
<td>File produced by C compiler (Freescale DSP only)</td>
</tr>
</tbody>
</table>

Figure E-3 File Header Flags

D.3.2 Optional Header

The COFF optional header ordinarily is used to hold system-dependent or runtime information. This allows different operating environments to store data that only that environment uses without forcing all COFF files to save space for that information. General utility programs can be made to work properly with any common object file. This is done by seeking past the optional header using the `f_opthdr` size field in the file header record.

The optional header in a Freescale DSP object file may contain two distinct types of information, depending upon how the file was generated. If the file is a relocatable object file it will have an optional header containing linker information. If the file is an absolute object file it will have an optional header containing runtime information. The runtime header is similar to standard COFF `a.out` optional header formats.

Figure E-4 shows the linker optional header. The module size field gives the size of the entire object module. The data size field reflects the size of the entire raw data block within the module. The `endstr` field points to an expression in the string table which originated with the assembler END directive (see Chapter 6); it indicates the starting address of the module. If this field is negative or zero there is no end expression. The logical section count is the count of sections in the object module created via the assembler SECTION directive (see Chapter 6). The counter count represents the number of COFF sections in the file (analogous to the file header `f_nscns` field). The relocation entry and line number counts hold the number of all relocation entries and line number records in the file. The
buffer and overlay counts give counts for each instance of a buffer or overlay in the module. The major version, minor version, and revision number fields reflect the assembler and linker versions to insure linker backward compatibility. The optional header flags hold special mode flags for the linker.

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Declaration</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-3</td>
<td>long int</td>
<td>modsize</td>
<td>Object module size</td>
</tr>
<tr>
<td>4-7</td>
<td>long int</td>
<td>datasize</td>
<td>Module raw data size</td>
</tr>
<tr>
<td>8-11</td>
<td>long int</td>
<td>endstr</td>
<td>End directive expression string</td>
</tr>
<tr>
<td>12-15</td>
<td>long int</td>
<td>secnt</td>
<td>Logical section count</td>
</tr>
<tr>
<td>16-19</td>
<td>long int</td>
<td>ctrcnt</td>
<td>Counter count</td>
</tr>
<tr>
<td>20-23</td>
<td>long int</td>
<td>relocnt</td>
<td>Relocation entry count</td>
</tr>
<tr>
<td>24-27</td>
<td>long int</td>
<td>lnocnt</td>
<td>Line number entry count</td>
</tr>
<tr>
<td>28-31</td>
<td>long int</td>
<td>bufcnt</td>
<td>Buffer count</td>
</tr>
<tr>
<td>32-35</td>
<td>long int</td>
<td>ovlcnt</td>
<td>Overlay count</td>
</tr>
<tr>
<td>36-39</td>
<td>long int</td>
<td>majver</td>
<td>Major version number</td>
</tr>
<tr>
<td>40-43</td>
<td>long int</td>
<td>minver</td>
<td>Minor version number</td>
</tr>
<tr>
<td>44-47</td>
<td>long int</td>
<td>revno</td>
<td>Revision number</td>
</tr>
<tr>
<td>48-51</td>
<td>long int</td>
<td>optflags</td>
<td>Optional header flags</td>
</tr>
</tbody>
</table>

Figure E-4 Freescale DSP Optional Link Header Format

Figure E-5 illustrates the runtime optional header. This header is similar to the standard COFF a.out header but there are differences. The magic number in this header is not the same as the magic number in the file header; this magic number is used indicate the file type to a host operating system. The magic number and version stamp fields currently are not used by the Freescale DSP tools and are set to zero. The text size field gives the size of all text-type data (executable code) in the object file. The data size field holds a count of all initialized data (apart from code) in the file. The uninitialized data size field is not used and is set to zero.
The program entry field represents the address given in the assembler **END** directive. The text start and data start values contain the low addresses for text and data segments, respectively. The text and data end values contain the high addresses for text and data segments, respectively. Note that addresses are expressed in terms of the C language typedef **CORE_ADDR**. A **CORE_ADDR** is a structure containing a long (4 byte) address and an enumeration type which classifies the address according to memory space (X, Y, L, P) and memory mapping (internal, external, etc.). See section D.4.1 for more information on the **CORE_ADDR** structure.

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Declaration</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-3</td>
<td>long int</td>
<td>magic</td>
<td>Magic number</td>
</tr>
<tr>
<td>4-7</td>
<td>long int</td>
<td>vstamp</td>
<td>Version stamp</td>
</tr>
<tr>
<td>8-11</td>
<td>long int</td>
<td>tsize</td>
<td>Size of text in words</td>
</tr>
<tr>
<td>12-15</td>
<td>long int</td>
<td>dsize</td>
<td>Size of data in words</td>
</tr>
<tr>
<td>16-19</td>
<td>long int</td>
<td>bsize</td>
<td>Size of uninitialized data in words</td>
</tr>
<tr>
<td>20-27</td>
<td>CORE_ADDR</td>
<td>entry</td>
<td>Program entry point</td>
</tr>
<tr>
<td>28-35</td>
<td>CORE_ADDR</td>
<td>text_start</td>
<td>Base address of text</td>
</tr>
<tr>
<td>36-43</td>
<td>CORE_ADDR</td>
<td>data_start</td>
<td>Base address of data</td>
</tr>
<tr>
<td>44-51</td>
<td>CORE_ADDR</td>
<td>text_end</td>
<td>End address of text</td>
</tr>
<tr>
<td>52-59</td>
<td>CORE_ADDR</td>
<td>data_end</td>
<td>End address of data</td>
</tr>
</tbody>
</table>

Figure E-5 Freescale DSP Optional Runtime Header Format

### D.3.3 Sections

A section is the smallest portion of an object file that is treated as one separate and distinct entity. Sections can accommodate program text, initialized and uninitialized data, and block data. COFF sections in DSP object files may be grouped under a logical section defined by the assembler **SECTION** directive (see Chapter 6).

It is a mistake to assume that every COFF file will have a specific number of sections, or to assume characteristics of sections such as their order, their location in the object file, or the address at which they are to be loaded. This information is available only after the
Section Headers

Every object file has a table of section headers to specify the layout of data within the file. The section header table consists of one entry for every section in the file. The information in the section header is described in Figure E-6.

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Declaration</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-7</td>
<td>char</td>
<td>s_name</td>
<td>Section name (null padded)</td>
</tr>
<tr>
<td>8-15</td>
<td>CORE_ADDR</td>
<td>s_paddr</td>
<td>Physical address</td>
</tr>
<tr>
<td>16-23</td>
<td>CORE_ADDR</td>
<td>s_vaddr</td>
<td>Virtual address</td>
</tr>
<tr>
<td>24-27</td>
<td>long int</td>
<td>s_size</td>
<td>Section size in words</td>
</tr>
<tr>
<td>28-31</td>
<td>long int</td>
<td>s_scnptr</td>
<td>File pointer to raw data</td>
</tr>
<tr>
<td>32-35</td>
<td>long int</td>
<td>s_relpTR</td>
<td>File pointer to relocation entries</td>
</tr>
<tr>
<td>36-39</td>
<td>long int</td>
<td>s_lnnoptr</td>
<td>File pointer to line number entries</td>
</tr>
<tr>
<td>40-43</td>
<td>unsigned long</td>
<td>s_nreloc</td>
<td>Number of relocation entries</td>
</tr>
<tr>
<td>44-47</td>
<td>unsigned long</td>
<td>s_nlnno</td>
<td>Number of line number entries</td>
</tr>
<tr>
<td>48-51</td>
<td>long int</td>
<td>s_flags</td>
<td>Section flags (see Figure E-7)</td>
</tr>
</tbody>
</table>

Figure E-6  Section Header Format

The section name is an 8-byte character array padded with null (zero) bytes if required. In Freescale relocatable object files section names may be longer than eight characters. In this case the convention used for long symbol names is followed where if the least significant four bytes of the section name field contain zeroes, the name is in the symbol table at the offset given by the most significant four bytes of the name field. See section D.3.4.1 for more information on the handling of long symbol names.

The physical address is the address where the section text or data will reside in memory. The address value depends upon whether the section is absolute or relocatable. If the section is absolute then the physical address is the actual address where the section will be loaded into memory. If the section is relocatable then the physical address is an offset...
from the start of the logical section (implicit or defined by the SECTION directive) in which the section is defined.

In most cases the virtual address is the same as the physical address. However, for block data sections in Freescale DSP object files the virtual address field holds the repeat count for the single raw data value associated with this section. For example, if the assembly language source file included a directive of the form BSC $400,$FFFF the $s_vaddr$ field would contain the value $400$, the $s_size$ field would be 1 (or 2 if in L memory), and the single raw data word associated with the section would be $FFFF$.

The section size is the count of raw data words associated with the section. This is in contrast to standard COFF section sizes which usually are given in bytes. Raw data words currently are stored in the object file as long (4-byte) integers independent of the target processor word size.

The file pointer fields are file byte offsets into the object file to the start of the current section raw data, relocation entries, and line number information. The counts of relocation and line number entries provide an upper bound for scanning these tables. The section flags comprise the section attributes and are described in Figure E-7.
Appendix D Freescale DSP Object File Format (COFF) - Object File Components

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>STYP_REG</td>
<td>$0000</td>
<td>Regular section</td>
</tr>
<tr>
<td>STYP_DSECT</td>
<td>$0001</td>
<td>Dummy section</td>
</tr>
<tr>
<td>STYP_NOLOAD</td>
<td>$0002</td>
<td>Noload section</td>
</tr>
<tr>
<td>STYP_GROUP</td>
<td>$0004</td>
<td>Grouped section</td>
</tr>
<tr>
<td>STYP_PAD</td>
<td>$0008</td>
<td>Padding section</td>
</tr>
<tr>
<td>STYP_COPY</td>
<td>$0010</td>
<td>Copy section</td>
</tr>
<tr>
<td>STYP_TEXT</td>
<td>$0020</td>
<td>Executable text section</td>
</tr>
<tr>
<td>STYP_DATA</td>
<td>$0040</td>
<td>Initialized data section</td>
</tr>
<tr>
<td>STYP_BSS</td>
<td>$0080</td>
<td>Uninitialized data section</td>
</tr>
<tr>
<td>STYP_BLOCK</td>
<td>$0400</td>
<td>Block data section</td>
</tr>
<tr>
<td>STYP_OVERLAY</td>
<td>$0800</td>
<td>Overlay section</td>
</tr>
<tr>
<td>STYP_MACRO</td>
<td>$1000</td>
<td>Macro section</td>
</tr>
</tbody>
</table>

Figure E-7  Section Header Flags

Text sections are reserved for code to be loaded into program memory (P space). Data sections hold initialized data, generated by assembler DC directives for example, bound for data (X, Y, L) memory. Bss sections are used for uninitialized blocks resulting from assembler DS and similar directives. Padding sections are generated to provide alignment when a modulo or reverse-carry buffer is declared. The block section attribute flags a block data section, described above. The overlay flag indicates the section is part of an overlay. Macro sections represent code and data generated during a macro expansion. Dummy sections are used internally by the assembler to mark empty sections after the first assembly pass. Empty sections may still appear in the object file if a symbol is associated with a section which contains no data. The noload, group, and copy attributes are not used at present.

D.3.3.2 Relocation Information

Object files have one relocation entry for each relocatable reference in the text or data. The relocation information consists of entries with the format described in Figure E-8.
Appendix D Freescale DSP Object File Format (COFF) - Object File Components

The address field represents the relocatable address within the section raw data where a modification is needed. In standard COFF the \texttt{r_symndx} field points to an entry in the symbol table corresponding to the reference requiring modification. The relocation type encodes how the raw data is to be changed to reflect the resolved symbol value.

In Freescale DSP COFF \texttt{r_symndx} is an offset into the string table which points to a relocation expression. The linker interprets this expression and updates the word at \texttt{r_vaddr} with the result of the expression evaluation. The relocation type is always zero. See section D.5, Object File Data Expression Format for more information on relocation expressions.

### D.3.3.3 Line Numbers

When the compiler or assembler is invoked with the \texttt{-G} debug option an entry is made in the object for every source line where a breakpoint can be inserted. It is then possible to reference source line numbers when using a debugger. The structure of an object file line entry is shown in Figure E-9.

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Declaration</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-3</td>
<td>long</td>
<td>l_symndx</td>
<td>Function name symbol table index</td>
</tr>
<tr>
<td>0-7</td>
<td>CORE_ADDR</td>
<td>l_paddr</td>
<td>Line number physical address</td>
</tr>
<tr>
<td>8-11</td>
<td>unsigned long</td>
<td>l_lnno</td>
<td>Source file line number</td>
</tr>
</tbody>
</table>

Figure E-9 Line Number Entry Format

All line numbers in a section are grouped by function as shown in Figure E-10. The first entry in a function grouping has line number 0 and has, in place of the physical address,
an index into the symbol table for the entry containing the function name. Subsequent entries have actual line numbers and addresses of the program text corresponding to the line numbers. The line number entries are relative to the beginning of the function, and appear in increasing order of address.

<table>
<thead>
<tr>
<th>Symbol index</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical address</td>
<td>Line number</td>
</tr>
<tr>
<td>Physical address</td>
<td>Line number</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>Symbol index</td>
<td>0</td>
</tr>
<tr>
<td>Physical address</td>
<td>Line number</td>
</tr>
<tr>
<td>Physical address</td>
<td>Line number</td>
</tr>
</tbody>
</table>

Figure E-10  Line Number Grouping

D.3.4 Symbol Table

The COFF symbol table serves a dual purpose: it provides resolution for symbolic references in relocation expressions during linking, and it establishes a framework for the handling of symbolic debug information. The symbol table consists of at least one fixed-length entry per symbol with some symbols followed by auxiliary entries of the same size.

Because of symbolic debugging requirements the order of symbols in the symbol table is very important. Whereas an individual symbol table entry can completely describe a single debugging entity, the entities exist within the framework of the source language that produced them. For example, symbol scoping and function blocks in C are represented by the appropriate ordering of begin-end block entries in the symbol table. Symbols in the symbol table appear in the sequence shown in Figure E-11.
The entry for each symbol is a structure that holds the symbol value, its type, and other information. There are symbol table entries used for relocation and linking and there are special symbols used only for debugging. The two kinds of entries are distinguished by combinations of field values in the symbol record. The structure of a symbol table entry is illustrated in Figure E-12.
### D.3.4.1 Symbol Name

The first eight bytes in the symbol table entry are a union of a character array and two longs. If the symbol name is seven characters or less, the null-padded symbol name is stored there. If the symbol name is longer than seven characters, then the entire symbol name is stored in the string table. In this case, the eight bytes contain two long integers: the first is zero and the second is the offset (relative to the beginning of the string table) of the name in the string table. Since there can be no symbols with a null name, the zeroes on the first four bytes serve to distinguish a symbol table entry with an offset from one with a name in the first eight bytes.

### D.3.4.2 Symbol Value

The symbol value is a union of a CORE_ADDR typedef and an array of two longs. If the symbol value is an address the contents will be stored as a CORE_ADDR structure with memory and mapping attributes. Otherwise the contents are stored in the `n_val` field. Whether the symbol value is an address or not depends on the storage class of the symbol. See section D.3.4.5 for more information on the relationship of symbol value and storage class.

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Declaration</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-7</td>
<td>char</td>
<td>n_name</td>
<td>Symbol name (null padded)</td>
</tr>
<tr>
<td>0-3</td>
<td>long int</td>
<td>n_zeroes</td>
<td>Zero in this field indicates name is in string table</td>
</tr>
<tr>
<td>4-7</td>
<td>long int</td>
<td>n_offset</td>
<td>Offset of name in string table</td>
</tr>
<tr>
<td>8-15</td>
<td>CORE_ADDR</td>
<td>n_address</td>
<td>Symbol address value</td>
</tr>
<tr>
<td>8-15</td>
<td>unsigned long</td>
<td>n_val[2]</td>
<td>Symbol value</td>
</tr>
<tr>
<td>16-19</td>
<td>long int</td>
<td>n_scnum</td>
<td>Symbol section number</td>
</tr>
<tr>
<td>20-23</td>
<td>unsigned long</td>
<td>n_type</td>
<td>Symbol basic and derived type</td>
</tr>
<tr>
<td>24-27</td>
<td>long int</td>
<td>n_sclass</td>
<td>Symbol storage class</td>
</tr>
<tr>
<td>28-31</td>
<td>long int</td>
<td>n_numaux</td>
<td>Number of auxiliary entries</td>
</tr>
</tbody>
</table>

Figure E-12 Symbol Table Entry Format
D.3.4.3 Section Number

The section number maps a symbol to its corresponding section in the object file (e.g. the section in which the symbol is defined). A special section number (-2) marks symbolic debugging symbols, including structure/union/enumeration tag names, typedefs, and the name of the file. A section number of -1 indicates that the symbol has a value but is not relocatable. Examples of absolute-valued symbols include automatic and register variables, function arguments, and end-of-structure symbols. A section number of 0 flags a relocatable external symbol that is not defined in the current file. Section numbers greater than zero correlate to the ordinal sequence of sections in the object file.

D.3.4.4 Symbol Type

The type field in the symbol table entry contains information about the basic and derived type for the symbol. This information is generated by the compiler and assembler only if the -G debug option is used. Each symbol has exactly one basic or fundamental type but can have more than one derived type. The type information is encoded as sets of bits in the field. Bits 0 through 3 hold one of the fundamental type values given in Figure E-13.
### Fundamental Types

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_NULL</td>
<td>0</td>
<td>Type not assigned</td>
</tr>
<tr>
<td>T_VOID</td>
<td>1</td>
<td>Void</td>
</tr>
<tr>
<td>T_CHAR</td>
<td>2</td>
<td>Character</td>
</tr>
<tr>
<td>T_SHORT</td>
<td>3</td>
<td>Short integer</td>
</tr>
<tr>
<td>T_INT</td>
<td>4</td>
<td>Integer</td>
</tr>
<tr>
<td>T_LONG</td>
<td>5</td>
<td>Long integer</td>
</tr>
<tr>
<td>T_FLOAT</td>
<td>6</td>
<td>Floating point</td>
</tr>
<tr>
<td>T_DOUBLE</td>
<td>7</td>
<td>Double word floating point</td>
</tr>
<tr>
<td>T_STRUCT</td>
<td>8</td>
<td>Structure</td>
</tr>
<tr>
<td>T_UNION</td>
<td>9</td>
<td>Union</td>
</tr>
<tr>
<td>T_ENUM</td>
<td>10</td>
<td>Enumeration</td>
</tr>
<tr>
<td>T_MOE</td>
<td>11</td>
<td>Member of enumeration</td>
</tr>
<tr>
<td>T_UCHAR</td>
<td>12</td>
<td>Unsigned character</td>
</tr>
<tr>
<td>T_USHORT</td>
<td>13</td>
<td>Unsigned short</td>
</tr>
<tr>
<td>T_UINT</td>
<td>14</td>
<td>Unsigned integer</td>
</tr>
<tr>
<td>T_ULONG</td>
<td>15</td>
<td>Unsigned long</td>
</tr>
</tbody>
</table>

**Figure E-13 Fundamental Types**

Bits 4 through 15 are arranged as six 2-bit subfields. These subfields represent levels of the derived types given in Figure E-14.
As an example of encoding fundamental and derived types, consider a function returning a pointer to a character. The fundamental type is character, giving bits 0-3 of the symbol type field the value 2. Bits 4-5 would hold a 2 for the derived type of function and bits 6-7 would contain a 1 for the pointer derived type. The value in the symbol entry type field would result in %01100010 binary, or $62 hexadecimal.

D.3.4.5 Symbol Storage Class

The symbol storage class indicates how a symbol will be used during execution or debugging. Some storage classes actually reflect how a symbol will be stored, e.g. as a register parameter. Other storage classes provide information for special symbols used in debugging, such as the beginning of blocks or the end of functions. Storage classes are outlined in Figure E-15.
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_EFCN</td>
<td>-1</td>
<td>Physical end of function</td>
</tr>
<tr>
<td>C_NULL</td>
<td>0</td>
<td>No storage class</td>
</tr>
<tr>
<td>C_AUTO</td>
<td>1</td>
<td>Automatic variable</td>
</tr>
<tr>
<td>C_EXT</td>
<td>2</td>
<td>External symbol</td>
</tr>
<tr>
<td>C_STAT</td>
<td>3</td>
<td>Static symbol</td>
</tr>
<tr>
<td>C_REG</td>
<td>4</td>
<td>Register variable</td>
</tr>
<tr>
<td>C_EXTDEF</td>
<td>5</td>
<td>External definition</td>
</tr>
<tr>
<td>C_LABEL</td>
<td>6</td>
<td>Label</td>
</tr>
<tr>
<td>C_ULABEL</td>
<td>7</td>
<td>Undefined label</td>
</tr>
<tr>
<td>C_MOS</td>
<td>8</td>
<td>Member of structure</td>
</tr>
<tr>
<td>C_ARG</td>
<td>9</td>
<td>Function argument</td>
</tr>
<tr>
<td>C_STRTAG</td>
<td>10</td>
<td>Structure tag</td>
</tr>
<tr>
<td>C_MOU</td>
<td>11</td>
<td>Member of union</td>
</tr>
<tr>
<td>C_UNTAG</td>
<td>12</td>
<td>Union tag</td>
</tr>
<tr>
<td>C_TPDEF</td>
<td>13</td>
<td>Type definition</td>
</tr>
<tr>
<td>C_USTATIC</td>
<td>14</td>
<td>Uninitialized static</td>
</tr>
<tr>
<td>C_ENTAG</td>
<td>15</td>
<td>Enumeration tag</td>
</tr>
<tr>
<td>C_MOE</td>
<td>16</td>
<td>Member of enumeration</td>
</tr>
<tr>
<td>C_REGPARAM</td>
<td>17</td>
<td>Register parameter</td>
</tr>
<tr>
<td>C_FIELD</td>
<td>18</td>
<td>Bit field</td>
</tr>
</tbody>
</table>

Figure E-15  Storage Classes
### Figure E-15  Storage Classes (continued)

The value of a symbol depends on its storage class. This relationship is summarized in Figure E-16.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_BLOCK</td>
<td>100</td>
<td>Beginning and end of block</td>
</tr>
<tr>
<td>C_FCN</td>
<td>101</td>
<td>Beginning and end of function</td>
</tr>
<tr>
<td>C_EOS</td>
<td>102</td>
<td>End of structure</td>
</tr>
<tr>
<td>C_FILE</td>
<td>103</td>
<td>C language source filename</td>
</tr>
<tr>
<td>C_LINE</td>
<td>104</td>
<td></td>
</tr>
<tr>
<td>C_ALIAS</td>
<td>105</td>
<td>Duplicated tag</td>
</tr>
<tr>
<td>C_HIDDEN</td>
<td>106</td>
<td></td>
</tr>
<tr>
<td>A_FILE</td>
<td>200</td>
<td>Assembly source filename</td>
</tr>
<tr>
<td>A_SECT</td>
<td>201</td>
<td>Beginning and end of section</td>
</tr>
<tr>
<td>A_BLOCK</td>
<td>202</td>
<td>Beginning/end of COFF section</td>
</tr>
<tr>
<td>A_MACRO</td>
<td>203</td>
<td>Macro expansion</td>
</tr>
<tr>
<td>A_GLOBAL</td>
<td>210</td>
<td>Global assembly language symbol</td>
</tr>
<tr>
<td>A_XDEF</td>
<td>211</td>
<td>XDEFed symbol</td>
</tr>
<tr>
<td>A_XREF</td>
<td>212</td>
<td>XREFed symbol</td>
</tr>
<tr>
<td>A_SLOCAL</td>
<td>213</td>
<td>Section local label</td>
</tr>
<tr>
<td>A_ULOCAL</td>
<td>214</td>
<td>Underscore local label</td>
</tr>
<tr>
<td>A_MLOCAL</td>
<td>215</td>
<td>Macro local label</td>
</tr>
<tr>
<td>Storage Class</td>
<td>Value</td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------</td>
<td></td>
</tr>
<tr>
<td>C_AUTO</td>
<td>Stack offset in words</td>
<td></td>
</tr>
<tr>
<td>C_EXT</td>
<td>Relocatable address</td>
<td></td>
</tr>
<tr>
<td>C_STAT</td>
<td>Relocatable address</td>
<td></td>
</tr>
<tr>
<td>C_REG</td>
<td>Register number</td>
<td></td>
</tr>
<tr>
<td>C_LABEL</td>
<td>Relocatable address</td>
<td></td>
</tr>
<tr>
<td>C_MOS</td>
<td>Offset in words</td>
<td></td>
</tr>
<tr>
<td>C_ARG</td>
<td>Stack offset in words</td>
<td></td>
</tr>
<tr>
<td>C_STRTAG</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>C_MOU</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>C_UNTAG</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>C_TPDEF</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>C_ENTAG</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>C_MOE</td>
<td>Enumeration value</td>
<td></td>
</tr>
<tr>
<td>C_REGPARAM</td>
<td>Register number</td>
<td></td>
</tr>
<tr>
<td>C_FIELD</td>
<td>Bit displacement</td>
<td></td>
</tr>
<tr>
<td>C_BLOCK</td>
<td>Relocatable address</td>
<td></td>
</tr>
<tr>
<td>C_FCN</td>
<td>Relocatable address</td>
<td></td>
</tr>
<tr>
<td>C_EOS</td>
<td>Size of structure in words</td>
<td></td>
</tr>
<tr>
<td>C_FILE</td>
<td>(see below)</td>
<td></td>
</tr>
<tr>
<td>C_ALIAS</td>
<td>Tag index</td>
<td></td>
</tr>
<tr>
<td>C_HIDDEN</td>
<td>Relocatable address</td>
<td></td>
</tr>
</tbody>
</table>

Figure E-16  Storage Class and Value
If a symbol has storage class C_FILE or A_FILE, the value of that symbol equals the symbol table entry index of the next C_FILE or A_FILE symbol. That is, the C_FILE and A_FILE entries form a one-way linked list in the symbol table. If there are no more C_FILE or A_FILE entries in the symbol table, the value of the symbol is the index of the first global symbol.

Relocatable symbols have a value equal to the relocatable address of that symbol. When the section is relocated by the linker, the value of these symbols changes.

### D.3.4.6 Auxiliary Entries

Every symbol table entry may have zero, one, or more auxiliary entries. These auxiliary entries are used to hold additional information about the primary symbol. The number of auxiliary entries associated with a given symbol can be determined by examining the `n_numaux` field of the main symbol entry.

An auxiliary symbol table entry contains the same number of bytes as its associated symbol table entry and is contiguous with the primary entry in the object file. Unlike primary symbol table entries, however, the format of an auxiliary entry depends on the type and storage class of the main symbol.
D.3.4.6.1 Filenames

The auxiliary table entry for a filename contains a 14-character array followed by an unsigned long integer. If the integer is zero then the filename is in the array. Otherwise it is in the string table at the offset given by the integer value. The \texttt{x_ftype} field indicates the memory space used for the stack in compiled modules.

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Declaration</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-13</td>
<td>char</td>
<td>x_fname</td>
<td>Source file name</td>
</tr>
<tr>
<td>14-17</td>
<td>unsigned long</td>
<td>x_foff</td>
<td>String table offset to file name</td>
</tr>
<tr>
<td>18-21</td>
<td>unsigned long</td>
<td>x_ftype</td>
<td>Memory space used by stack</td>
</tr>
</tbody>
</table>

Figure E-17 Filename Symbol Auxiliary Entry

D.3.4.6.2 Sections

Section auxiliary entries have the format shown in Figure E-18. This information is analogous to selected fields in the corresponding section header. If the object file is relocatable a section symbol entry will have a second auxiliary entry with the format shown in Figure E-19.

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Declaration</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-3</td>
<td>long int</td>
<td>x_scnlen</td>
<td>Section length</td>
</tr>
<tr>
<td>4-7</td>
<td>unsigned long</td>
<td>x_nreloc</td>
<td>Number of relocation entries</td>
</tr>
<tr>
<td>8-11</td>
<td>unsigned long</td>
<td>x_nlinno</td>
<td>Number of line numbers</td>
</tr>
<tr>
<td>12-31</td>
<td>-</td>
<td>-</td>
<td>Unused (zero filled)</td>
</tr>
</tbody>
</table>

Figure E-18 Section Symbol Auxiliary Entry
The logical section number is the ordinal related to a `SECTION` directive in the assembler source file. The relocation section number usually is the same as the logical section number, but may be different if the logical section is static within an enclosing section. The memory mapping is an alternate encoding of the `CORE_ADDR` information in the section header. Section type flags indicate whether this COFF section represents a buffer or overlay block. If the current COFF section is a buffer or overlay block a third auxiliary entry is produced. The layout of that entry is shown in Figure E-19.

Buffers and overlays are mutually exclusive so their respective fields share storage space in the object file. The buffer section number is really the buffer instance count in this file.
Buffer type is either modulo or reverse carry. The buffer limit gives the upper bound for the buffer size even though the block may contain less initialized data than this limit suggests. The overlay memory structure gives the runtime memory attributes for this block. The overlay section number is really the overlay instance count in this file. The overlay origin expression is the expression given for the runtime counter in the assembler `ORG` directive (see Chapter 6).

D.3.4.6.3 Tag Names

Auxiliary entries for C language structure and union tag names have the format described in Figure E-21. Note that in Freescale DSP COFF the size of the associated structure or union is in words as opposed to bytes as in standard COFF. The `x_endndx` field is used to create a linked list of tag name entries through the symbol table.

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Declaration</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-7</td>
<td>-</td>
<td>-</td>
<td>Unused (zero filled)</td>
</tr>
<tr>
<td>8-11</td>
<td>unsigned long</td>
<td>x_size</td>
<td>Size of structure, union, or enumeration in words</td>
</tr>
<tr>
<td>12-15</td>
<td>-</td>
<td>-</td>
<td>Unused (zero filled)</td>
</tr>
<tr>
<td>16-19</td>
<td>long int</td>
<td>x_endndx</td>
<td>Index of next structure, union, or enumeration entry</td>
</tr>
<tr>
<td>20-31</td>
<td>-</td>
<td>-</td>
<td>Unused (zero filled)</td>
</tr>
</tbody>
</table>

Figure E-21  Tag Name Symbol Auxiliary Entry

D.3.4.6.4 End of Structures

The format for C language end-of-structure auxiliary entries is given in Figure E-22. Note that the size of the structure, union, or enumeration is given in words rather than bytes. The tag index holds the symbol table index for the tag record associated with this structure.
Appendix D Freescale DSP Object File Format (COFF) - Object File Components

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Declaration</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-3</td>
<td>long int</td>
<td>x_tagndx</td>
<td>Tag index</td>
</tr>
<tr>
<td>4-7</td>
<td>-</td>
<td>-</td>
<td>Unused (zero filled)</td>
</tr>
<tr>
<td>8-11</td>
<td>unsigned long</td>
<td>x_size</td>
<td>Size of structure, union, or enumeration in words</td>
</tr>
<tr>
<td>12-31</td>
<td>-</td>
<td>-</td>
<td>Unused (zero filled)</td>
</tr>
</tbody>
</table>

Figure E-22 End of Structure Auxiliary Entry

D.3.4.6.5 Functions

Function auxiliary entries have the format shown in Figure E-23. Note that the size of the function is given in words rather than bytes. The function tag index holds the symbol table index to the begin-function symbol for this function. The x_endndx field points to the next function symbol table entry. The x_innopr field contains a byte offset pointer within the object file to the line number entry that signals the start of this function (see section D.3.3.3, Line Numbers, for more information).

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Declaration</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-3</td>
<td>long int</td>
<td>x_tagndx</td>
<td>Tag index</td>
</tr>
<tr>
<td>4-7</td>
<td>long int</td>
<td>x_fsize</td>
<td>Size of function in words</td>
</tr>
<tr>
<td>8-11</td>
<td>-</td>
<td>-</td>
<td>Unused (zero filled)</td>
</tr>
<tr>
<td>12-15</td>
<td>long int</td>
<td>x_innopr</td>
<td>File pointer to line number entry</td>
</tr>
<tr>
<td>16-19</td>
<td>long int</td>
<td>x_endndx</td>
<td>Index of next function entry</td>
</tr>
<tr>
<td>20-31</td>
<td>-</td>
<td>-</td>
<td>Unused (zero filled)</td>
</tr>
</tbody>
</table>

Figure E-23 Function Symbol Auxiliary Entry
D.3.4.6.6 Arrays

The format for C language array auxiliary entries is given in Figure E-24. The tag index contains the offset to the next array symbol in the symbol table. The line number field gives the source file line number for the array declaration.

![Figure E-24 Array Symbol Auxiliary Entry](image)

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Declaration</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-3</td>
<td>long int</td>
<td>x_tagndx</td>
<td>Tag index</td>
</tr>
<tr>
<td>4-7</td>
<td>unsigned long</td>
<td>x_lnno</td>
<td>Line number of array declaration</td>
</tr>
<tr>
<td>8-11</td>
<td>unsigned long</td>
<td>x_size</td>
<td>Size of array</td>
</tr>
<tr>
<td>12-15</td>
<td>unsigned long</td>
<td>x_dimen[0]</td>
<td>First array dimension</td>
</tr>
<tr>
<td>16-19</td>
<td>unsigned long</td>
<td>x_dimen[1]</td>
<td>Second array dimension</td>
</tr>
<tr>
<td>20-23</td>
<td>unsigned long</td>
<td>x_dimen[2]</td>
<td>Third array dimension</td>
</tr>
<tr>
<td>24-27</td>
<td>unsigned long</td>
<td>x_dimen[3]</td>
<td>Fourth array dimension</td>
</tr>
<tr>
<td>28-31</td>
<td></td>
<td></td>
<td>Unused (zero filled)</td>
</tr>
</tbody>
</table>

D.3.4.6.7 End of Blocks and Functions

The format for C language symbol entries for the end of blocks and functions is given in Figure E-25. Only the source file line number for the end of the block or function is stored.

![Figure E-25 End of Block or Function Auxiliary Entry](image)

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Declaration</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-3</td>
<td></td>
<td></td>
<td>Unused (zero filled)</td>
</tr>
<tr>
<td>4-7</td>
<td>unsigned long</td>
<td>x_Inno</td>
<td>Source file line number</td>
</tr>
<tr>
<td>8-31</td>
<td></td>
<td></td>
<td>Unused (zero filled)</td>
</tr>
</tbody>
</table>
D.3.4.6.8 Beginning of Blocks and Functions

The format for C language symbol entries for the beginning of blocks and functions is described in Figure E-26. The source file line number is retained. The \texttt{x\_endndx} provides a link to the next beginning of block or function symbol in the symbol table.

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Declaration</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-3</td>
<td>-</td>
<td>-</td>
<td>Unused (zero filled)</td>
</tr>
<tr>
<td>4-7</td>
<td>unsigned long</td>
<td>x_lnno</td>
<td>Source file line number</td>
</tr>
<tr>
<td>8-15</td>
<td>-</td>
<td>-</td>
<td>Unused (zero filled)</td>
</tr>
<tr>
<td>16-19</td>
<td>long int</td>
<td>x_endndx</td>
<td>Index of next beginning of block or function</td>
</tr>
<tr>
<td>20-23</td>
<td>unsigned long</td>
<td>x_type</td>
<td>Function prologue/epilogue index</td>
</tr>
<tr>
<td>24-31</td>
<td>-</td>
<td>-</td>
<td>Unused (zero filled)</td>
</tr>
</tbody>
</table>

Figure E-26 Beginning of Block or Function Auxiliary Entry

D.3.4.6.9 Structure, Union, and Enumeration Names

The format for auxiliary entries related to structure, union, and enumeration names is given in Figure E-27. The tag index is used to access the tag symbol record that describes this structure. Note that in Freescale DSP COFF the size of the associated structure or union is in words as opposed to bytes as in standard COFF.
Object File Comments

Object file comments are stored in a single COFF symbol table entry. The symbol name field contains the special comment string .cmt. The n_address field points to the comment text as an offset into the string table. Comments generated automatically by the assembler or via the IDENT directive have an n_scnum field with value of -1. Comments produced with the COBJ directive generally have the section number of the COFF section in which they reside. The type and storage class fields are both zero.

String Table

Symbol and section names longer than seven characters and comment text are stored contiguously in the string table with each string delimited by a zero byte. The first four bytes represent the size of the string table in bytes; offsets into the string table, therefore, are always greater than or equal to 4. An empty string table has a length field with value zero.

Differences in DSP Object Format and Standard COFF

Freescale DSP COFF is substantially the same as generic COFF and usage of format elements is similar. However, the original COFF specification did not envision aspects of machine architecture which the Freescale DSP family possesses. Moreover, standard COFF encompasses a file format which is quite adaptable among host processors, but is not necessarily portable among those hosts. It is straightforward enough to adapt COFF to a new host machine, but the intent is that the derived host format will be recognized and executed only on that target host. For Freescale DSP COFF the format had to be extended for cross-development such that a given object file would be usable on all targeted host systems. The following sections outline the differences and changes between standard COFF and Freescale DSP COFF.

---

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Declaration</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-3</td>
<td>long int</td>
<td>x_tagndx</td>
<td>Tag index</td>
</tr>
<tr>
<td>4-7</td>
<td>-</td>
<td>-</td>
<td>Unused (zero filled)</td>
</tr>
<tr>
<td>8-11</td>
<td>unsigned long</td>
<td>x_size</td>
<td>Size of structure, union, or enumeration in words</td>
</tr>
<tr>
<td>12-31</td>
<td>-</td>
<td>-</td>
<td>Unused (zero filled)</td>
</tr>
</tbody>
</table>
D.4.1 Multiple Memory Spaces

Standard COFF has no built-in mechanism for accommodating multiple memory spaces. It does handle the notion of separate text and data sections, and a possible extension would have been to define section types for the new memory areas. This quickly becomes unwieldy when mapping information (internal, external, port A/B) is considered as well.

The solution was to extend addressing information to include the memory and mapping with the address value itself. This is done by defining a C language typedef called CORE_ADDR which holds both the memory and mapping data along with the memory address. For any address context in the COFF file a CORE_ADDR is used rather than, for example, an unsigned long. A description of the CORE_ADDR format is shown in Figure E-28.

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Declaration</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-3</td>
<td>long</td>
<td>w0.l</td>
<td>Memory address</td>
</tr>
<tr>
<td>4-7</td>
<td>enum</td>
<td>w1.mape</td>
<td>Memory mapping</td>
</tr>
</tbody>
</table>

Figure E-28 CORE_ADDR Format

The enumeration values for the memory mapping field are shown in order in Figure E-29.
D.4.2 Object File Transportability

There are many different structure definitions in the COFF specification. These definitions consist of fields comprised of varying C data types. These data types are recognized by any reasonable C compiler, but their characteristics and sizes may change from machine to machine. This is acceptable if the COFF files are to be used only on a particular machine architecture. But if COFF files are produced on one machine to be used on another several problems may arise. One is that since the data fields can vary in size there could be alignment problems when accessing structures or individual fields. Another issue is byte ordering between machines. Given an arbitrary byte stream, some machines store the bytes in a word starting at the least significant bit (LSB) end of the word, while others store bytes starting at the most significant bit (MSB) end of the word.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Value</th>
<th>Mnemonic</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory_map_p</td>
<td>0</td>
<td>memory_map_xa</td>
<td>16</td>
</tr>
<tr>
<td>memory_map_x</td>
<td>1</td>
<td>memory_map_xb</td>
<td>17</td>
</tr>
<tr>
<td>memory_map_y</td>
<td>2</td>
<td>memory_map_xe</td>
<td>18</td>
</tr>
<tr>
<td>memory_map_l</td>
<td>3</td>
<td>memory_map_xi</td>
<td>19</td>
</tr>
<tr>
<td>memory_map_none</td>
<td>4</td>
<td>memory_map_xr</td>
<td>20</td>
</tr>
<tr>
<td>memory_map_laa</td>
<td>5</td>
<td>memory_map_ya</td>
<td>21</td>
</tr>
<tr>
<td>memory_map_lab</td>
<td>6</td>
<td>memory_map_yb</td>
<td>22</td>
</tr>
<tr>
<td>memory_map_lba</td>
<td>7</td>
<td>memory_map_ye</td>
<td>23</td>
</tr>
<tr>
<td>memory_map_lbb</td>
<td>8</td>
<td>memory_map_yi</td>
<td>24</td>
</tr>
<tr>
<td>memory_map_le</td>
<td>9</td>
<td>memory_map_yr</td>
<td>25</td>
</tr>
<tr>
<td>memory_map_li</td>
<td>10</td>
<td>memory_map_pt</td>
<td>26</td>
</tr>
<tr>
<td>memory_map_pa</td>
<td>11</td>
<td>memory_map_pf</td>
<td>27</td>
</tr>
<tr>
<td>memory_map_pb</td>
<td>12</td>
<td>memory_map_emi</td>
<td>28</td>
</tr>
<tr>
<td>memory_map_pe</td>
<td>13</td>
<td>memory_map_e0-63</td>
<td>29-92</td>
</tr>
<tr>
<td>memory_map_pi</td>
<td>14</td>
<td>memory_map_error</td>
<td>666666</td>
</tr>
<tr>
<td>memory_map_pr</td>
<td>15</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure E-29 Memory Mapping Enumerations
The Freescale DSP version of COFF addresses these potential problems by normalizing the object file. Normalization occurs in a number of ways. All structure and union elements are converted to long values, and raw data is stored in 4-byte quantities independent of the word size of the target processor. In some cases this wastes space in the object file and in memory but it was considered worth the price for transportability among supported hosts. Also it is not a completely portable solution by any means (e.g. for machines with larger than 4-byte word sizes).

The byte ordering issue was dealt with by establishing a baseline ordering, providing compliance for foreign hosts with conversion code. This introduces overhead logic on machines that do not support the baseline word order but again it was seen as a reasonable trade-off to insure transportability of object files among development environments. Note that byte swapping logic only comes into play for fields that are not byte-atomic, such as integer fields. Character arrays in structures, for example, should not have their bytes exchanged.

The byte ordering for Freescale DSP COFF is shown in Figure E-30. It adheres to what sometimes is called the big-endian approach to byte and word ordering.

<table>
<thead>
<tr>
<th>Addr n</th>
<th>Addr n+1</th>
<th>Addr n+2</th>
<th>Addr n+3</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSB</td>
<td>MSB - 1</td>
<td>LSB + 1</td>
<td>LSB</td>
</tr>
</tbody>
</table>

Figure E-30 Freescale DSP COFF Byte Ordering

D.4.3 Structure Size Fields

In some of the COFF data structures there is a size field which gives the size of a block in the target processor environment. For example, there are several symbol table auxiliary entries that specify the size of a structure or union for debug purposes. In standard COFF these sizes ordinarily are in bytes but in Freescale DSP COFF they are given in words unless otherwise indicated. The use of word sizes for debug entities should be distinguished from file pointer offset values in the object file. File pointers are indeed byte offsets within the object file that are used by utilities to process information in the object file itself.

D.4.4 Relocation Information

In standard COFF the r_symndx field of any given relocation record points to an entry in the symbol table corresponding to a symbol reference requiring modification. When the standard COFF linker performs symbol resolution, pairing symbol definitions with matching references, it updates the relocation entry to point to the symbol definition and dis-
cards the reference symbol. When the relocation entries are processed, the resolved symbol value is used to modify the raw data indicated by the relocation entry at r_vaddr.

In Freescale DSP COFF r_symndx is an offset into the string table which points to a relocation expression. The linker interprets this expression and updates the entire word at r_vaddr with the result of the expression evaluation. The relocation type is always zero. See section D.5, Object File Data Expression Format for more information on relocation expressions.

D.4.5 Block Data Sections

Generic COFF does not make allowance for a block data section. A block data section results from use of the assembler BSC directive, where a large block of memory is initialized with a single value. Block data sections are handled in Freescale DSP COFF by making special use of the section s_vaddr field and adding an informative flag.

In most cases the section virtual address is the same as the physical address. However, for block data sections in Freescale DSP object files the virtual address field holds the repeat count for the single raw data value associated with the section. For example, if the assembly language source file included a directive of the form BSC $400,$FFFF the s_vaddr field would contain the value $400, the s_size field would be 1 (or 2 if in L memory), and the single raw data word associated with the section would be $FFFF. In addition, the STYP_BLOCK flag is set in the section s_flags field.

D.4.6 Other Extensions

If the object file is relocatable there are extra structures which the assembler and linker generate to support special constructs such as logical sections, buffers, and overlays. The optional link file header contains information which the linker requires; it is described in section D.3.2. Every symbol table entry for a section in a relocatable file has an extra auxiliary entry described in section D.3.4.6.2.

One special DSP COFF structure not documented elsewhere is the comment symbol. A comment symbol table entry is emitted either indirectly via the assembler IDENT directive or directly with the COBJ directive (see Chapter 6). A comment symbol table entry may be identified by a symbol name of .cmt and a type and storage class of zero. The value field of a comment symbol holds the offset into the string table of the comment text. The section number for a comment symbol produced with the IDENT directive is always -1. Comment symbols generated with the COBJ directive have the section number of the section where the COBJ directive appears in the source file. Comment symbols have no auxiliary entry.

D.5 Object File Data Expression Format

Object file data expressions are used in data relocation records to represent values to be loaded into memory. An expression is a combination of symbols, constants, operators,
and parentheses. Expressions may contain user-defined labels, integers, floating point numbers, or literal strings. An object file data expression generally follows the guidelines of assembler expressions, except that functions are not supported (e.g. they must be evaluated at assembly time), and operators are provided for linker-specific operations. Also, floating point terms found in these expressions are converted to binary values.

### D.5.1 Data Expression Generation

Link file data expressions are generated when external or relocatable operands are encountered during assembly or incremental link processing. In most cases the operand expression is copied verbatim from the source and embellished with link evaluation control constructs. For example, consider the source line below:

```
MOVE #FOO,R0
```

The DSP96000 assembler produces the following encoding for this line in the object file:

```
$3A8D2000 {FOO}@0#0
```

Since the symbol FOO is not known to the assembler it generates a two-word instruction and places a relocation reference to the expression in the position of the second instruction word. The braces ({} ) indicate that this is a user expression that should adhere to certain integrity constraints such as those governing absolute and relative terms. Otherwise the braces are treated much like parentheses. The at sign (@) is a binary operator indicating the memory space of the left operand by the right. The pound sign (#) is a binary operator signifying the size in bits of the left operand by the right. More information on these special operators and their operands is given below.

Here is another example of data expression generation:

```
JCLR #1,X:LOC,LABEL
```

For this conditional jump the assembler produces the following object file code:

```
((S02A00481&~(~(~0<<8)<<12))I(({LOC}@1#8&~(~0<<8))<<12))  {LABEL}@0#0
```

The first expression is evaluated such that the relative address LOC, resolved at link time, is shifted and masked into the middle eight bits of the base instruction word ($02A00481). The expression could have been more complex if the bit number was an external reference. The relative value of the symbol LABEL occupies the second instruction word.

### D.5.2 Data Expression Interpretation

Object file data expressions are similar to standard assembler expressions which generally follow the rules of algebra and boolean arithmetic. They are written using infix notation in conjunction with unary and binary operators and parentheses. There are also extensions to the usual set of assembler arithmetic and grouping operators. These are control constructs that assist the linker in determining the size, type, and characteristics of an expression operand.
D.5.2.1 User Expression - { ... }

The curly braces ({ }) delimit a user expression within a data expression. A user expression is that part of a data expression that was written by the programmer and not generated by the assembler or linker as part of its control requirements. It is useful to isolate the user expression in order to check for relationships among absolute and relative terms. In all other respects the curly braces behave like parentheses.

D.5.2.2 Relocatable Expression - [ ... ]

The square brackets ([ ]) are used to enclose a relocatable expression. The value contained in the square brackets is an offset from the base of the current section. Usually this grouping operator is placed around the value of an assembler local label (underscore label) since these symbols do not migrate to the link file.

D.5.2.3 Memory Space Operator - @

The at sign (@) is a binary operator that checks the memory space compatibility of the left operand based on the value of the right operand. The compatibility check is made based on the matrix outlined in section 3.3, Expression Memory Space Attribute.

D.5.2.4 Bit Size Operator - #

The pound sign (#) is a binary operator used to verify the size in bits of the left operand given the value of the right operand. The following bit sizes and operand type correspondences are defined:

-16 - 16-bit signed short immediate or offset
-15 - 15-bit signed short PC-relative address
-7  - 7-bit signed short immediate or offset
-6  - 6-bit signed short PC-relative address
-5  - 5-bit signed short offset
-1  - Negated immediate shift
0   - DSP word size immediate or absolute
1   - Immediate shift
5       -  5-bit short absolute
6       -  6-bit short immediate or absolute
7       -  7-bit short immediate, absolute, or offset
8       -  8-bit short immediate or absolute
12      -  12-bit short immediate or absolute
19      -  19-bit short immediate
85,86,87       -  5,6,7-bit I/O short absolute

D.5.2.5 Memory Attribute Operator - :

The colon (:) is used to assign a memory space and counter encoded in the right operand to the left operand. The low sixteen bits of the right operand contain the counter designator for the left operand. The high sixteen bits contain the memory space designator for the left operand. The value here corresponds to the memory space values given for the memory space operator (@) described above.

D.5.2.6 Line Number Operator - !

The exclamation point (!) is a binary operator that associates the source file line number of the left operand to the value of the right operand. The left operand is a decimal value representing the source file line number. The right operand is an arbitrary relocation expression. The line number operator assists the linker in correlating source line numbers to expression terms which could be evaluated erroneously at link time, e.g. forced operands which do not fit into the instruction word after relocation.

D.5.2.7 BFxxx Instruction Mask Function - @FBF()

The @FBF() function is generated by the assembler when the mask operand for a bitfield instruction is external. The assembler passes a relocation expression to the function as an argument. The linker evaluates the expression in parentheses and adjusts the associated instruction accordingly.

D.5.2.8 Local Relocatable Reference Function - @LRF()

The @LRF() function attempts to encapsulate more detail about a relocatable expression than is generated with other operators such as @ and ::. Its main purpose is to provide sufficient information for evaluating local relocatable expressions that are referenced outside a defining context. For example, the assembler might produce an @LRF() function for a reference to an underscore label that is outside its defining section. The function arguments consist of the original relocatable expression, the memory space/mapping value, the location counter designation, the defining and relocation section numbers, and any buffer or overlay sequence numbers associated with the expression.
D.5.2.9 Alternate Encoding Function - @ENC()

The @ENC() provides two encoding expressions for the same instruction. This is useful in cases where the unknown value of an external operand can affect which expression is used for the final encoding. The function takes four arguments. The first argument to the function is the external operand expression. The second argument is the absolute value beyond which the second encoding expression is used over the first. The third argument is the first encoding expression and the last argument is the second encoding expression.
Appendix E Instruction Set Information

E.1 DSP56300 Information

The Freescale DSP56300 refers to a family of high-speed, low power programmable CMOS processors. The DSP56300 supports 24-bit signed fixed-point fractional arithmetic.

E.1.1 Instruction Set Summary

DSP56300 instructions can be grouped by function into six types:

1. Arithmetic instructions
2. Logical instructions
3. Bit manipulation instructions
4. Loop instructions
5. Move instructions
6. Program control instructions

E.1.1.1 Arithmetic Instructions

The DSP56300 instructions used for arithmetic operations are:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS</td>
<td>Absolute value*</td>
</tr>
<tr>
<td>ADC</td>
<td>Add long with carry*</td>
</tr>
<tr>
<td>ADD</td>
<td>Add*</td>
</tr>
<tr>
<td>ADDL</td>
<td>Shift left then add*</td>
</tr>
<tr>
<td>ADDR</td>
<td>Shift right then add*</td>
</tr>
<tr>
<td>ASL</td>
<td>Arithmetic shift accumulator left*</td>
</tr>
<tr>
<td>ASR</td>
<td>Arithmetic shift accumulator right*</td>
</tr>
<tr>
<td>CLR</td>
<td>Clear accumulator*</td>
</tr>
<tr>
<td>CMP</td>
<td>Compare*</td>
</tr>
<tr>
<td>CMPM</td>
<td>Compare magnitude*</td>
</tr>
</tbody>
</table>

*Instruction allows parallel data move.
E.1.1.2 Logical Instructions

The DSP56300 instructions used for logical operations are:

- **AND**: Logical AND*
- **ANDI**: AND Immediate with control register
- **EOR**: Logical exclusive OR*
- **LSL**: Logical shift accumulator left*
- **LSR**: Logical shift accumulator right*
- **NOT**: Logical complement on accumulator*
- **OR**: Logical inclusive OR*
- **ORI**: OR immediate with control register
- **ROL**: Rotate accumulator left*
- **ROL**: Rotate accumulator right*

E.1.1.3 Bit Manipulation Instructions

The DSP56300 instructions used for bit manipulation are:

- **BCHG**: Bit test and change
- **BCLR**: Bit test and clear
- **BSET**: Bit test and set
- **BSET**: Bit test on memory

*Instruction allows parallel data move.*
E.1.1.4 Loop Instructions

The DSP56300 instructions used for loop operations are:

- **DO** - Start hardware loop
- **ENDDO** - Exit from hardware loop

E.1.1.5 Move Instructions

The DSP56300 instructions used for move operations are:

- **LUA** - Load updated address
- **MOVE** - Move data*
- **MOVEC** - Move control register
- **MOVEM** - Move program memory
- **MOVEP** - Move peripheral data

E.1.1.6 Program Control Instructions

The DSP56300 instructions used for program control are:

- **DEBUG** - Enter debug mode
- **DEBUGcc** - Enter debug mode conditionally
- **ILLEGAL** - Illegal instruction interrupt
- **Jcc** - Jump conditionally
- **JCLR** - Jump if bit clear
- **JMP** - Jump
- **JScc** - Jump to subroutine conditionally
- **JSCLR** - Jump to subroutine if bit clear
- **JSET** - Jump if bit set
- **JSSET** - Jump to subroutine if bit set
- **JSR** - Jump to subroutine
- **NOP** - No operation
- **REP** - Repeat next instruction
- **RESET** - Reset on-chip peripheral devices
- **RTI** - Return from interrupt
- **RTS** - Return from subroutine
- **STOP** - Stop processing (low power standby)
- **SWI** - Software interrupt
- **WAIT** - Wait for interrupt (low power standby)

*Instruction allows parallel data move.
E.1.2  Register Names and Usage

The following DSP56300 register names, in either upper or lower case, cannot be used as symbol names in an assembly language source file:

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, A, AB</td>
<td></td>
</tr>
<tr>
<td>X0, A0, BA</td>
<td></td>
</tr>
<tr>
<td>X1, A1, A10</td>
<td></td>
</tr>
<tr>
<td>Y, B, B10</td>
<td></td>
</tr>
<tr>
<td>Y0, B0, A2</td>
<td></td>
</tr>
<tr>
<td>Y1, B1, B2</td>
<td></td>
</tr>
<tr>
<td>R0, N0, M0, MR</td>
<td></td>
</tr>
<tr>
<td>R1, N1, M1, CCR</td>
<td></td>
</tr>
<tr>
<td>R2, N2, M2, SR</td>
<td></td>
</tr>
<tr>
<td>R3, N3, M3, LC</td>
<td></td>
</tr>
<tr>
<td>R4, N4, M4, LA</td>
<td></td>
</tr>
<tr>
<td>R5, N5, M5, SSH</td>
<td></td>
</tr>
<tr>
<td>R6, N6, M6, SSL</td>
<td></td>
</tr>
<tr>
<td>R7, N7, M7, OMR</td>
<td></td>
</tr>
</tbody>
</table>

The following DSP56300 registers are used by the assembler in structured control statement processing (Chapter 7):

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>A, X0, Y0</td>
<td></td>
</tr>
</tbody>
</table>

E.1.3  Condition Code Mnemonics

Following are the DSP56300 condition code mnemonics which correspond to the conditional instructions based on the CCR condition codes. These tests may be used in an operand comparison expression within a structured control statement (Chapter 7).

- **<CC>** - carry clear
- **<CS>** - carry set
- **<EC>** - extension clear
- **<EQ>** - equal
- **<ES>** - extension set
- **<GE>** - greater or equal
- **<GT>** - greater than
- **<HS>** - higher or same
- **<LC>** - limit clear
- **<LE>** - less or equal
- **<LO>** - lower
- **<LS>** - limit set
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;LT&gt;</td>
<td>less than</td>
</tr>
<tr>
<td>&lt;MI&gt;</td>
<td>minus</td>
</tr>
<tr>
<td>&lt;NE&gt;</td>
<td>not equal</td>
</tr>
<tr>
<td>&lt;NN&gt;</td>
<td>not normalized</td>
</tr>
<tr>
<td>&lt;NR&gt;</td>
<td>normalized</td>
</tr>
<tr>
<td>&lt;PL&gt;</td>
<td>plus</td>
</tr>
</tbody>
</table>
Appendix E Instruction Set Information - DSP56300 Information
INDEX

— A —

Absolute expression 17
Absolute mode 2, 9
Address assignment 42
Addressing
  I/O short 85
  immediate 88
  long 87
  long immediate 90
  short 86
  short immediate 89
Assembler
  command line 1
  device 267
  error 196
  fatal error 226
  installation 1
  mode 132
  operation 1
  option 6, 135
  output 15
  processing 8
  verbose 7
  warning 142, 188
Assembly language 1

— B —

Bootstrap mode 54
Buffer 9
  address 91
  circular 47
Index

end 112

— C —

Checksum 137, 138
COFF 231, 257
Command line 1
  input 4
Command line option 2
  -A 2
  -B 3
  -D 3
  -EA 4
  -EW 4
  -F 4
  -G 4
  -I 5
  -L 5
  -M 6
  -O 6
  -P 6
  -Q 6
  -V 7
  -Z 7
Comment 97
  delimiter 72
  object file 96, 257, 261
  unreported 73
Comment field 15
Conditional assembly 57, 65, 124, 137
Constant 19
  binary 19
  decimal 19
  define 98, 99
  hexadecimal 19
  storage 93
  string 20
Cycle count 137, 138

— D —

Data block 261
Data hiding 38
Data transfer field 14
Debug 4
Directive 67
  .BREAK 70, 168, 183
  .CONTINUE 70, 169, 183
  .ELSE 70, 183
  .ENDF 70, 183
  .ENDI 70, 183
  .ENDL 70, 183
  .ENDW 70, 183
  .FOR 70, 170, 183
  .IF 70, 171, 183
  .LOOP 70, 172, 183
  .REPEAT 70, 173, 183
  .UNTIL 70, 183
  .WHILE 70, 173, 183
assembly control 68, 181
BADDR 69, 91, 182
BSB 69, 92, 182
BSC 69, 93, 182
BSM 69, 94, 182
BUFFER 69, 95, 182
COBJ 70, 96, 183
COMMENT 68, 97, 181
data definition 69, 182
DC 69, 98, 182
DCB 69, 99, 182
DEFINE 3, 68, 79, 100, 138, 181
dS 69, 101, 182
DSM 69, 102, 182
DSR 69, 103, 182
DUP 65, 70, 104, 183
DUPA 65, 70, 106, 183
DUPC 65, 70, 108, 183
DUPF 65, 70, 109, 183
END 68, 111, 181
ENDBUF 69, 112, 182
ENDIF 70, 113, 183
Index

ENDM  58, 70, 114, 183
ENDSEC  69, 115, 182
EQU  69, 116, 182
EXITM  70, 117, 183
FAIL  68, 118, 181
FORCE  68, 119, 181
GLOBAL  69, 120, 182
GSET  69, 121, 182
HIMEM  68, 122, 181
IDENT  70, 123, 183
IF  65, 70, 124, 183
in loop  138
INCLUDE  68, 125, 181
LIST  69, 126, 182
listing control  69, 182
LOCAL  69, 127, 182
LOMEM  68, 128, 181
LSTCOL  69, 129, 182
MACLIB  6, 58, 70, 130, 183
MACRO  58, 70, 131, 183
macro  70, 183
MODE  41, 68, 132, 181
MSG  68, 133, 181
NOLIST  69, 134, 182
object file  70, 183
OPT  6, 69, 135, 182
ORG  43, 68, 143, 181
PAGE  69, 147, 182
PMACRO  70, 149, 183
PRCTL  69, 150, 182
RADIX  68, 151, 181
RDIRECT  57, 68, 152, 181
SCSJMP  68, 153, 181
SCSREG  68, 154, 181
SECTION  69, 155, 182
SET  39, 69, 158, 182
STITLE  69, 159, 182
structured control  70, 183
symbol definition  69, 182
SYMOBJ  70, 160, 183
TABS  69, 161, 182
TITLE  69, 162, 182
UNDEF  68, 163, 181
WARN  68, 164, 181
XDEF  38, 69, 165, 182
XREF  38, 69, 166, 182
DSP56300  267
condition code  270
instruction set  267
register  270
DSPASMOPT  2

— E —

Environment variable  2
Error
command line  186
fatal  226
output  4
Expression  17
absolute  17
address  137
compound  176
condition code  174
constant  19
formatting  176
function  23
internal representation  19
object file  261
operand comparison  175
operator  20
radix  151
relative  17
simple  174

— F —

File
command line  4
include  5, 125
input  11
listing  5, 15, 139
Index

macro 6
object 3, 231
output 7
source 7, 11
Function 23, 81
ABS 24, 26
ACS 24, 26
ARG 25, 26
ASN 24, 27
AT2 24, 27
ATN 24, 27
CCC 26, 27
CEL 24, 27
CHK 26
CNT 25, 28
COH 24, 28
conversion 25
COS 24, 28
CTR 26, 28
CVF 25, 29
CVI 25, 29
CVS 25, 29
DEF 26, 29
EXP 26, 29
FLD 25, 30
FLR 24, 30
FRC 25, 30
INT 26, 30
L10 24, 30
LCV 26, 31
LEN 25, 31
LFR 25, 31
LNG 25, 31
LOG 24, 32
LST 26, 32
LUN 25, 32
MAC 25, 32
macro 25
mathematical 24
MAX 24, 32

MIN 24, 33
miscellaneous 26
MSP 26, 33
MXP 25, 33
POS 25
POW 24, 33, 34
REL 26, 34
RND 24, 34
RVB 25, 34
SCP 25, 34
SGN 24, 35
SIN 24, 35
SNH 24, 35
SQT 24, 35
string 25
TAN 24, 35
TNH 24, 36
UNF 25, 36
XPN 24, 36

— I —
Include file 5, 125

— L —
Label 13
local 13, 59, 64, 139, 142
Label field 13
Line continuation 74
Line number 240
Listing file 5, 15, 139
format 129, 138, 141, 147, 161
sub-title 159
title 162
Load address 8
Load location counter 9
Load memory space 8, 9
Local label 13
macro 13
Location counter 82, 143
— M —

Macro 57
  call 60, 139
  comment 138
  definition 58, 131, 139
  directive 131
  end 114
  exit 117
  expansion 60, 139
  file 6
  library 58, 130, 139
  purge 149

Macro argument
  concatenation operator 61, 74
  local label override 64
  local label override operator 78
  return hex value operator 63, 77
  return value operator 62, 76
  string operator 63

Memory
  limit 122, 128
  utilization 15, 139

Memory space 8, 139, 143
  attribute 17

— O —

Object file 3
  auxiliary entry 250
  comment 96, 257, 261
  data expression 261
  differences 257
  file header 233
  format 231
  identification 123
  line number 240
  optional header 234
  relocation 239, 260
  section 236
  section number 244
  storage class 246

structure 231
structure size 260
symbol 141, 160
symbol name 243
symbol table 241
symbol type 244
symbol value 243
transportability 259

Operand field 14
Operation field 14
Operator 20
  arithmetic 21
  bitwise 22
  logical 22
  precedence 23
  relational 22
  shift 21
  unary 20

Option
  AE 136, 137
  assembler operation 137
  CC 137
  CEX 136, 137
  CK 137
  CL 136, 137
  CM 137, 138
  CONST 137, 138
  CONTC 138
  CONTCK 137, 138
  CRE 136, 138
  DEX 136, 138
  DLD 137, 138
  DXL 136, 138
  FC 135, 138
  FF 135, 138
  FM 135, 138
  GL 137, 138
  GS 137, 138
  HDR 136, 138
  IC 136, 138
Index

IL  136, 139  NOCL  136, 139  NOSI  140
INTR 137, 139  NOSCL  140
LB  137, 139  NOSI  140
LDB 137, 139  NOU  141
listing format  135  NOUI  141
LOC 136, 139  NOW  141
MC  136, 139  NS  136, 141
MD  136, 139  PP  135, 141
message  136  PS  137, 141
MEX 136, 139  PSM  137
MI  137, 139  RC  135, 141
MSW 136, 139  reporting  136
MU  136, 139  RP  137, 141
NL  136, 139  RSV  137
NOAE 139  S  136, 141
NOCC 139  SCL  136, 141
NOCEX 139  SCO  136, 141
NOCK 139  SI  137, 141
NOCL 139  SO  136, 141
NOCM 140  SVO  137
symbol  136
NODEX 140  U  136, 142
NODLD 140  UR  136, 142
NODXL 140  W  136, 142
NOFC 140  WEX  142
NOFF 140  XLL  136, 142
NOFM 140  XR  136, 142
OGS 140  Overlay  8, 45, 51
NOHDR 140  bootstrap  54
NINTR 140  Program counter  82, 143
NOMC 140  Relative expression  17
NOMD 140  Relative mode  9
NOMEX 140  Relocation  239, 260
NOMI 140  Runtime address  8
NOMMSW 140  Runtime location counter  9
NONL 140  Runtime memory space  8, 9
NONS 140
NOPP 140
NOPS 140
NORC 140
NORP 140
Index

— S —

Section 37, 155
  block data 261
  data hiding 38
  end 115
  fragmented 40
  global 39, 120, 138, 155
  header 237
  local 39, 127, 155
  macro 40
  nested 40, 141
  relocation 41
  static 42, 138, 155
  symbol 38
Source file 7, 11
  end 111
  format 12
String 11
  concatenation 12, 83, 84
  delimiter 79
  packed 141
  position 33
  substring 12
Symbol 11
  case 138
  cross-reference 138
  equate 116, 138
  global 138
  listing 141
  set 121, 158
  strip 7
  undefined 142

— W —

Warning 142, 188