CodeWarrior Development Studio for StarCore 3900FP DSP Architectures Assembler Reference Manual

Document Number: CWSCASMREF
Rev. 10.9.0, 06/2015

freescale™

Contents

Section number **Title** **Page**

Chapter 1
Introduction

1.1 Assembler.. 11

1.2 Software Development Flow.. 11

Chapter 2
StarCore Assembler

2.1 Starting the Assembler...13

2.2 Command-Line Options...14

2.2.1 Using an Environment Variable...18

2.2.2 Reading Input from an Argument File...19

2.2.3 Generating an Object File.. 20

2.2.4 Adding Debug Information...20

2.2.5 Redirecting the Source Listing...21

2.2.6 Controlling Assembler Messages...21

2.2.7 Searching Additional Directories...22

2.2.8 Defining Substitution Strings...22

2.2.9 Using OPT Options on the Command Line.. 23

2.2.10 Counting the core stalls..23

2.2.11 Specifying a Target Architecture.. 24

2.2.12 Specifying Endian Mode..24

2.2.13 Checking Programming Rules... 25

2.2.13.1 Code Examples.. 26

2.2.13.2 Data Analysis Terms...28

2.2.13.3 Data Analysis Limitations..28

2.2.13.4 Initialization File..29

2.3 Assembler Processing.. 30

2.4 Source Statements...31

2.4.1 Label Field... 32

Section number	Title	Page

2.4.2 Operation Field.......... 32

2.4.3 Operand Field.......... 33

2.4.4 Comment Field.......... 33

2.4.5 Variable Length Execution Sets.......... 33

2.4.6 Symbol Names.......... 34

2.4.7 Symbol Labels.......... 35

2.4.8 Strings.......... 35

2.5 Source Listing.......... 36

2.5.1 Source Listing Example.......... 37

Chapter 3
Expressions

3.1 Absolute and Relative Expressions.......... 39

3.2 Expression Memory Space Attributes.......... 40

3.3 Internal Expression Representation.......... 41

3.4 Constants.......... 41

3.4.1 Numeric Constants.......... 41

3.4.2 String Constants.......... 41

3.5 Operators.......... 42

3.6 Operator Precedence.......... 44

3.7 Functions.......... 45

3.7.1 ABS Absolute Value.......... 47

3.7.2 ACS Arc Cosine.......... 47

3.7.3 ARG Macro Argument.......... 48

3.7.4 ASN Arc Sine.......... 48

3.7.5 AT2 Arc Tangent.......... 49

3.7.6 ATN Arc Tangent.......... 49

3.7.7 BIGENDIAN Endian Mode Check.......... 49

3.7.8 CCC Cumulative Cycle Count.......... 50

3.7.9 CEL Ceiling.......... 50

Section number	Title	Page

3.7.10 CHK Instruction/Data Checksum........ 51

3.7.11 CNT Macro Argument Count........ 51

3.7.12 COH Hyperbolic Cosine........ 52

3.7.13 COS Cosine........ 52

3.7.14 CTR Location Counter Number........ 53

3.7.15 CVF Convert Integer to Floating Point........ 53

3.7.16 CVI Convert Floating Point to Integer........ 54

3.7.17 CVS Convert Memory Space........ 54

3.7.18 DEF Defined Symbol........ 55

3.7.19 EXP Expression Check........ 55

3.7.20 FLD Shift and Mask........ 56

3.7.21 FLR Floor........ 56

3.7.22 FRC Convert Floating Point to Fractional........ 57

3.7.23 INT Integer Check........ 57

3.7.24 L10 Log Base 10........ 58

3.7.25 LCV Location Counter Value........ 58

3.7.26 LEN String Length........ 59

3.7.27 LFR Convert Floating Point to Long Fractional........ 59

3.7.28 LNG Concatenate to Double Word........ 60

3.7.29 LOG Natural Logarithm........ 60

3.7.30 LST LIST Directive Flag Value........ 61

3.7.31 LUN Convert Long Fractional to Floating Point........ 61

3.7.32 MAC Macro Definition........ 61

3.7.33 MAX Maximum Value........ 62

3.7.34 MIN Minimum Value........ 62

3.7.35 MSP Memory Space........ 63

3.7.36 MXP Macro Expansion........ 63

3.7.37 POS Position of Substring........ 63

3.7.38 POW Raise to a Power........ 64

3.7.39 REL Relative Mode.......... 65

3.7.40 RND Random Value.......... 65

3.7.41 RVB Reverse Bits in Field.......... 65

3.7.42 SCP Compare Strings.......... 66

3.7.43 SGN Return Sign.......... 66

3.7.44 SIN Sine.......... 66

3.7.45 SNH Hyperbolic Sine.......... 67

3.7.46 SQT Square Root.......... 67

3.7.47 TAN Tangent.......... 68

3.7.48 TNH Hyperbolic Tangent.......... 68

3.7.49 UNF Convert Fractional to Floating Point.......... 69

3.7.50 XPN Exponential Function.......... 69

Chapter 4
Software Project Management

4.1 Sections.......... 71

4.1.1 Section Names.......... 72

4.1.2 Nested and Fragmented Sections.......... 73

4.1.3 Sections and Symbols.......... 73

4.1.4 Macros and DEFINE Symbols within Sections.......... 74

4.2 Sections and Relocation.......... 74

4.3 Address Assignment.......... 75

4.4 Overlays.......... 75

4.4.1 Overlay Manager.......... 77

4.4.2 Overlay Example.......... 78

4.5 Multi-Programmer Environment Example.......... 81

4.5.1 Method 1: Absolute Mode.......... 82

4.5.2 Method 2: Relative Mode.......... 83

Chapter 5
Assembler Directives

5.1 Significant Characters....85

5.2 Directive List....86

5.3 Descriptions....88

5.3.1 ; Start Comment....89

5.3.2 ;; Start Unreported Comment....89

5.3.3 \ Continue Line....90

5.3.4 \ Concatenate Macro Argument....90

5.3.5 ? Substitute Macro Value....91

5.3.6 % Substitute Macro Hex Value....92

5.3.7 ^ Override Macro Local Label....93

5.3.8 " Delimit Macro String....93

5.3.9 " Expand DEFINE Quoted String....94

5.3.10 @ Start Function....95

5.3.11 * Substitute Location Counter....95

5.3.12 ++ Concatenate Strings....95

5.3.13 [ ] Delimit Substring....95

5.3.14 [ ] Group Instructions....96

5.3.15 < Force Short Addressing....96

5.3.16 > Force Long Addressing....97

5.3.17 # Use Immediate Addressing....98

5.3.18 #< Force Immediate Short Addressing....98

5.3.19 #> Force Immediate Long Addressing....99

5.3.20 ALIGN Align Location Counter....99

5.3.21 BADDR Set Buffer Address....100

5.3.22 BSB Allocate Bit-Reverse Buffer....101

5.3.23 BSC Allocate Constant Storage Block....102

5.3.24 BUFFER Start Buffer....102

Section number	Title	Page

5.3.25 COMMENT Start Comment Lines....104

5.3.26 DC Define Constant....104

5.3.27 DCB Define Constant Byte....105

5.3.28 DCL Define Constant Long....106

5.3.29 DCLL Define Constant Long Long....107

5.3.30 DEFINE Define Substitution String....108

5.3.31 DS Define Storage....109

5.3.32 DSR Define Reverse-Carry Storage....109

5.3.33 DUP Duplicate Source Lines....110

5.3.34 DUPA Duplicate Sequence with Arguments....111

5.3.35 DUPC Duplicate Sequence with Characters....112

5.3.36 DUPF Duplicate Sequence in Loop....113

5.3.37 ELSE Start Alternative Conditional Assembly....114

5.3.38 END End of Source Program....115

5.3.39 ENDBUF End Buffer....116

5.3.40 ENDIF End Conditional Assembly....116

5.3.41 ENDM End Macro Definition....117

5.3.42 ENDSEC End Section....117

5.3.43 EQU Equate Symbol to Value....118

5.3.44 EXITM Exit Macro....118

5.3.45 FAIL Issue Programmer Error Message....119

5.3.46 FALIGN Align with Fetch-Set....120

5.3.47 GLOBAL Declare Global Section Symbol....121

5.3.48 GSET Set Global Symbol to Value....121

5.3.49 HIMEM Set High Memory Bounds....122

5.3.50 IF Start Conditional Assembly....123

5.3.51 INCLUDE Include Secondary File....124

5.3.52 LIST List Assembly....125

5.3.53 LOMEM Set Low Memory Bounds....125

Section number | Title | Page

5.3.54 MACLIB Specify Macro Library........ 126

5.3.55 MACRO Define Macro........ 127

5.3.56 MSG Issue Programmer Message........ 128

5.3.57 MULTIDEF Allow Multiple Definitions........ 129

5.3.58 NOLIST Stop Assembly Listing........ 129

5.3.59 NOTE Include Note........ 130

5.3.60 OPT Set Assembler Options........ 130

5.3.61 ORG Initialize Memory Space and Location Counters........ 136

5.3.62 PAGE Advance Page or Size Page........ 137

5.3.63 PMACRO Purge Macro Definition........ 138

5.3.64 PRCTL Send Control String to Printer........ 139

5.3.65 SECFLAGS Set ELF Section Flags........ 140

5.3.66 SECTION Start Section........ 141

5.3.67 SECTYPE Set ELF Section Type........ 143

5.3.68 SET Set Symbol to Value........ 144

5.3.69 SIZE Set Symbol Size........ 145

5.3.70 STITLE Initialize Program Subtitle........ 146

5.3.71 TITLE Initialize Program Title........ 146

5.3.72 TYPE Set Symbol Type........ 147

5.3.73 UNDEF Undefine DEFINE Symbol........ 148

5.3.74 WARN Issue Programmer Warning........ 149

5.4 Pragmas........ 149

5.4.1 SECTYPE........ 150

5.4.2 STACK_EFFECT........ 150

Chapter 6
Macros and Conditional Assembly

6.1 Defining Macro........ 153

6.1.1 Calling a Macro........ 154

6.1.2 Macro Expansions........ 155

Section number	Title	Page

6.1.3 Macro Libraries....156

6.1.4 Dummy Argument Operators....156

6.1.5 Macro Directives....156

6.2 Conditional Assembly....157

Chapter 1
Introduction

The StarCore assembly language tools consist of an assembler, a linker, an archiver, and several utilities. This manual explains the assembly language tools for the StarCore DSP cores.

In this chapter:

- Assembler
- Software Development Flow

1.1 Assembler

The StarCore Assembler converts handwritten or compiler-generated StarCore assembly code into ELF object files.

The assembler provides:

- Expression evaluation involving numeric constants, string constants, operators, and built-in functions
- Modular programming involving sections
- Macros that allow variable arguments
- Conditional assembly
- Debug information of code written in assemble language
- Assembly code source files

Chapters 2 through 6 provide a complete explanation of the assembler.

1.2 Software Development Flow

This topic describes software development flow for StarCore Assembler.

The following diagram illustrates the software development flow, showing the inputs and outputs of each stage.

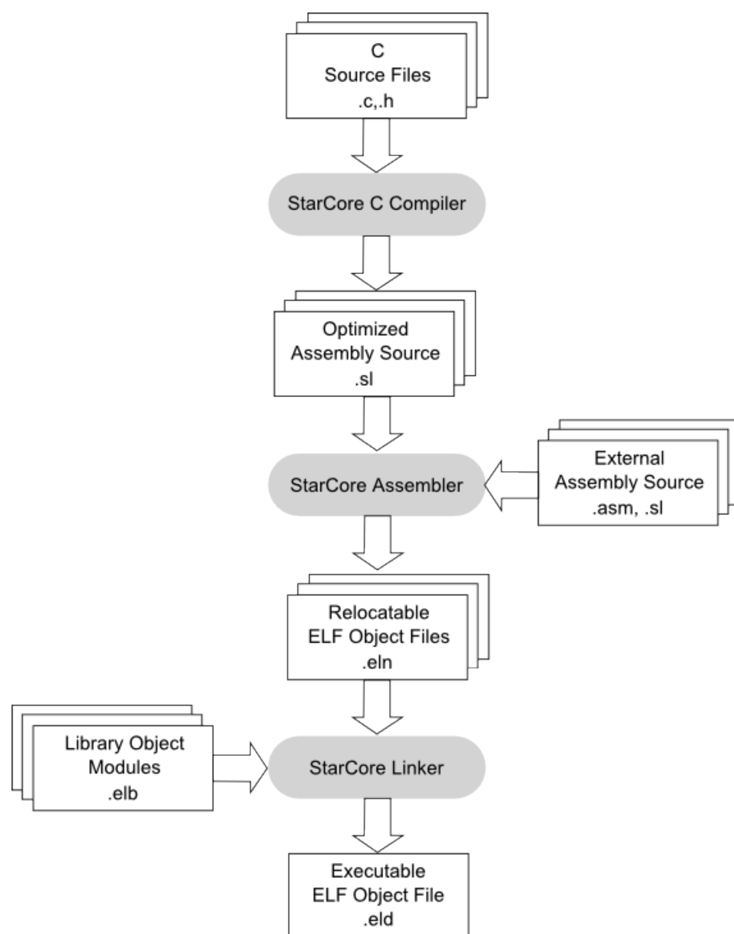


Figure 1-1. StarCore Development Tools

The StarCore assembler architecture is compiler-friendly, so you can combine it with a compiler that generates exceptionally compact code. In this manner, you can write applications in C, yet achieve code density and high performance comparable to that of hand-coded assembly programs.

Compiler options let you specify appropriate settings, development tools, processing stages, and processing options. Then, when you submit C source files to the compiler, its shell automatically advances the source files through compilation, assembly, and linking, to produce an executable program.

You submit hand-coded assembly language files to the assembler, which transforms the assembly code into ELF object code. The assembler then submits this object code and any object modules to the linker, which combines them into a single, executable program.

You may load this executable program into the simulator for execution and evaluation.

Chapter 2
StarCore Assembler

This section describes StarCore assembler and its features.

The StarCore Assembler translates hand-written or compiler-generated StarCore assembly language programs into machine language. This assembler uses executable and linking format (ELF) for object files.

The assembler supports these features:

- Expression evaluation using numeric constants, string constants, operators, and built-in functions
- Modular programming using sections
- Macros that allow variable arguments
- Conditional assembly
- Debug information of code written in assemble language
- Assembly code source files

In this chapter:

- Starting the Assembler
- Command-Line Options
- Assembler Processing
- Source Statements
- Source Listing

2.1 Starting the Assembler

This topic describes how to start the assembler in command-line mode.

Use this command to start the assembler:

```
scasm [option ...] file
```

where:

`option`

One or more of the optionsTable 2-1 table lists.

`file`

The assembly source file.

If a file's name does not include an extension, the assembler tries to open the file without an extension. If that is not successful, the assembler appends `.asm` to the name, and again tries to open the file.

NOTE

The assembler does not generate an object file unless the command includes the `-b` option.

This example command starts the assembler, assembles source file `corr.asm`, outputs a source listing to the standard output, and generates relocatable object file `corr.eln`. Except for the `-b` option, this command tells the assembler to use all default settings.

```
scasm -b corr.asm
```

2.2 Command-Line Options

This topic lists and describes the command-line options for StarCore assembler.

The table below summarizes assembler options; later sections of this chapter explain these option in more detail. Note that:

- Assembler options are not case sensitive.
- Certain options may appear more than once on the command line, as their descriptions explain.
- Options with one argument must have space in between otherwise no space is required with options that may or may not have arguments, or have variable number of arguments. Following are few of the examples:
  - -archsc3900fp (or -archsc3850) is wrong, the correct syntax is -arch sc3900fp
  - -u a3 is wrong, the correct syntax is -ua3
  - -b u0 is wrong, the correct syntax is -bu0

NOTE

All options in the below table and OPT Set Assembler Options are valid with the `-Xasm` pass through option of the `scc` command line.

Table 2-1. StarCore Assembler Options

Option	Description	See Section
`-a`	Specifies absolute mode instead of the default relative mode. With the `-b` option, generates an executable object file.	Generating an Object File
`-arch <arch>`	Specifies the assembly architecture for the file. The arch parameter values include: sc3900fp, b4460, and b4860 This option is synonymous with the command `sc3900fp`|`b4460`|`b4860`.	Specifying a Target Architecture
`-b[objfile]`	Generates an object file, assigning the specified name.	Generating an Object File
`-c`	Specifies compiler generated code, making `-ocs` and `-q` the default options. • Turns on restriction checking that disables programming rule A3. • Order matters for `-c`, `-s`, and `-u` options: assembler processes them from left to right. • Suppresses asm banner • Debug-information is not generated by the asm (accepted by compiler)	
`-dsymbolstring`	Defines substitution strings to be used on all source lines; equivalent to the `DEFINE` directive.	Defining Substitution Strings

Table continues on the next page...

Table 2-1. StarCore Assembler Options (continued)

Option	Description	See Section
`-ea errfile`	Appends the standard error output stream to the specified file.	Controlling Assembler Messages
`-ew errfile`	Writes the standard error output stream to the specified file, overwriting any previous contents. A space is mandatory before the file name.	Controlling Assembler Messages
`-fargfile`	Reads options and file names from the specified file, appending them to the command line.	
`-g`	Adds debug information to the object file; valid only with the `-b` option.	Adding Debug Information
`-ipathname`	Adds the specified directory to the standard search paths; repeatable multiple times. The assembler searches directories in their command-line order.	Searching Additional Directories
`-l[lstfile]`	Generates the listing file to the specified file.	Redirecting the Source Listing
`-mdirectory`	Specifies the directory that contains macro definitions; repeatable multiple times. The assembler searches directories in their command-line order. Equivalent to the `MACLIB` directive.	Searching Additional Directories
`-oopt[,opt...]`	Designates assembler options; commas without spaces must separate multiple options, as in example `-ofa,svo`. Equivalent to the `OPT` directive; valid arguments are any `OPT`-directive options, which OPT Set Assembler Options explains.	Using OPT Options on the Command Line
`-q`	Specifies to suppress the assembler banner.	Controlling Assembler Messages

Table continues on the next page...

Table 2-1. StarCore Assembler Options (continued)

Option	Description	See Section
`-s{all | none|strict|id[,id...]}`	Enables checks of programming rules (formerly called restriction checking). Commas without spaces must separate multiple arguments. • -snone suppresses implicit checking (equivalent to -uall). • -sall turns on all restriction checking. • -sstrict turns on restriction checking that could generate error messages (but not warnings). • Order matters for -s, -c, and -u options: assembler processes them from left to right.	Checking Programming Rules
`-u{all|none|id[,id...]}`	Inhibits restriction checking for the specified restrictions, as in `-ua1,a2` or `-uall`. • `-uall` is equivalent to `-snone`. • `-unone` is equivalent to `-sall`. • Order matters for `-u`, `-c`, and `-s` options: assembler processes them from left to right.	
`-version`	Displays assembler banner, then exits.	
`-W=[no]<keyword>`	Enables/disables warnings or remarks: • `-W=falign` - displays FALIGN remarks (equivalent to `-ofa`)	

Table 2-1. StarCore Assembler Options

Option	Description	See Section
	• `-W=remarks` - displays remarks (equivalent to `-or`) • `-W=warnings` - displays warnings (equivalent to `-ow`) • `-W=nofalign` - disables FALIGN remarks display (equivalent to `-onofa`) • `-W=noremarks` - disables remarks display (equivalent to `-onor`) • `-W=nowarnings` - disables warnings display (equivalent to `-onow`)	

2.2.1 Using an Environment Variable

If you use command-line options regularly, you may assign them to the environment variable DSPASMOPT. Before processing any options, the assembler adds this variable's text to the existing command line.

To define DSPASMOPT:

1. In the below table find the command line and environment file appropriate for your operating system. (The environment file is in the directory that `$SC100_HOME` defines.)
2. Enter the command line in the environment file.
    a. Any option of Table 2-1 table is a valid `option` parameter value.
    b. Separate multiple option values with spaces.
    c. Start each option value with a hyphen.

3. When you are done, re-execute the environment file.

Table 2-2. DSPASMOPT Command Lines

Operating System	Command Line	Environment File
UNIX: Bourne shell (sh, ksh, bash)	`DSPASMOPT="-option ..."export DSPASMOPT`	`env.sh`
UNIX: C shell (csh, tcsh)	`setenv DSPASMOPT "-option ..."`	`env.csh`
Windows	`set DSPASMOPT=-option ...`	`env.bat`

For example, if the DSPASMOPT definition in the `env.sh` file is:

```
DSPASMOPT="-b -l"
export DSPASMOPT
```

Then each time you invoke the assembler, it adds the `-b` and `-l` options to the command line. The command `scasm corr.asm` becomes `scasm -l -b corr.asm`.

2.2.2 Reading Input from an Argument File

The `-fargfile` option instructs the assembler to read command-line input from the specified argument file. This option is a method for passing command-line input from such an argument file to the assembler.

The `argfile` parameter value can include an optional pathname. You may repeat this option multiple times.

For example, this command invokes the assembler, telling it to read arguments from the file `asmopts`:

```
scasm -fasmopts -q filter.asm
```

An argument file is a text file containing a list of options, arguments, file names - even the `-f` option itself. Within the argument file, a space, blank, tab, or newline character must separate each file or option. Use semicolons to include comments.

Argument-file contents can be as simple as this example:

```
-b -l
-sa1,gg4
```

2.2.3 Generating an Object File

The assembler generates a relocatable object file only if the command line includes the `-b` option. If the command line includes both the `-a` and `-b` options, the assembler instead generates an executable object file.

`-a`

With the `-b` option, generates an executable object file.

`-b[ file]`

Generates an object file, assigning the specified name; the file name may include an optional pathname. Using a hyphen in place of a file name sends the object file to the standard output.

This option overwrites any file that has the same name. If this option does not include a file name, the assembler uses the next option in the command line as the file name.

This example assembles files `main.asm` and `fft.asm` into the executable object file `filter.eld`:

```
scasm -a -bfilter.eld main.asm fft.asm
```

2.2.4 Adding Debug Information

To add debug information to the object file, use the `-g` option:

`-g`

Adds these debugging sections to the object file: `.debug_abbrev`, `.debug_aranges`, `.debug_info`, `.debug_macinfo`, `.debug_loc`, and `.debug_line`. Produces debug information for all global symbols, including EQUs. Wherever necessary for debugging, inserts local symbols (named F_MemAllocArea_[section_name]_[pc]).

This option is valid only with the `-b` option.

Accordingly, for an assembly file from the compiler, your command line should include the `-c` option, to suppress assembly source-level debug information. But for a manually written assembly file, your command line should *not* include the `-c` option.

If the assembly file includes an overlay or union section, the assembler appends that section's name to the names of debug sections. For consistent debug information, modules that contain debug information and an overlay/union section must not include other text sections.

2.2.5 Redirecting the Source Listing

Per the default setting, the assembler sends a source listing to the standard output. To save the source listing to a file, use the `-l` option.

`-l[ file]`

Redirects the source listing to the specified file; the file name may include an optional pathname.

This option overwrites any file that has the same name. The parameter specifying the file name is optional. If this option does not include a name, the assembler uses the name of the first source file in the command line, with extension `.lst`.

This first example assembles files `filter.asm` and `gaus.asm` into the single, relocatable object file `filter.eln`, then redirects the source listing to file `filter.lst`:

```
scasm -b -lfilter.lst filter.asm gaus.asm
```

The second example inhibits the source listing, by specifying IL (inhibit listing) as an argument for the -o option. (Another way to inhibit the source listing is specifying IL as an argument to the OPT directive in the assembly source file.)

```
scasm -b -oil filter.asm gaus.asm
```

2.2.6 Controlling Assembler Messages

To redirect the standard error output stream to a file, and to control the level of messages the assembler displays, use the `-ea`, `-ew`, or `-q` options.

`-ea file`

Appends the standard error output stream (`stderr`) to the specified file. A space is required between `-ea` and the file name.

`-ew file`

Writes the standard error output stream (`stderr`) to the specified file, overwriting the file if it already exists. A space is required between `-ew` and the file name.

`-q`

Specifies to suppress the assembler banner.

This example:

- Assembles the files `filter.asm` and `gaus.asm` into the relocatable object file `filter.eln`,
- Redirects the standard error output stream to the file `errors`, and
- Redirects the source listing to the file `filter.lst`.

```
scasm -b -ew errors -lfilter.lst filter.asm gaus.asm
```

2.2.7 Searching Additional Directories

To add directories to the assembler's standard search paths, use the `-i` or `-m` options.

`-ipathname`

Adds the specified directory to the search path for INCLUDE files.

`-mpathname`

Adds the specified directory to the search path for macro definitions. This option is equivalent to the MACLIB directive.

You may repeat either of these directives multiple times. The assembler searches directories in their command-line order.

These examples add directory `sctools/fftlib` to the search path. (The first example is for a UNIX environment, the second for a Windows environment).

```
scasm -m/sctools/fftlib trans.asm
```

```
scasm -ic:\sctools\fftlib filter.asm
```

2.2.8 Defining Substitution Strings

To define substitution strings, use the `-d` option:

`-d` *`symbol string`*

Directs the assembler to replace every occurrence of symbol in the source file with the specified string. A space must precede the string. If the string contains spaces, single or double quotes must enclose the string.

You can repeat the `-d` *symbol string* sequence multiple times.

This example substitutes the string `1` for all occurrences of `BIG_ENDIAN` in the source file `vit.asm`.

```
scasm -b -dB_END '1' -obe vit.asm
```

Another way to define substitution strings is using the DEFINE directive in the source file.

2.2.9 Using OPT Options on the Command Line

To use any OPT-directive options on the command line, use the `-o` option.

`-o` *`opt[,opt...]`*

Directs the assembler to use the specified OPT-directive options. Commas without spaces must separate multiple options.

This example tells the assembler to include the `MD` and `MEX` options - that is, to include macro definitions and macro expansions in the source listing:

```
scasm -b -l -omd,mex corr.asm
```

2.2.10 Counting the core stalls

To instruct the assembler to output information regarding the core stalls, use the `-ostalls` option.

For example:

```
scasm -arch b4860 -l -ostalls input.asm
```

2.2.11 Specifying a Target Architecture

The valid arguments are:

- sc3900fp
- b4460
- b4860

The following table list the special symbols the assembler automatically defines for an ELF object file. These symbols relate to the architecture that command line specifies.

Table 2-3. Assembler Symbols

Architecture	Defines
SC3900FP/B4460/B4860	_SC3900_, __SC3900__

This first example uses the `-c` command-line option to invoke the assembler for the SC3900FP DSP core:

```
scasm -b -l  -dMY_DEF '1' main.asm
```

This second example uses the OPT directive to specify the core architecture:

```
      opt cex,mex
      page 132,42,0,0,0
LAB1  macro args
      ...
```

2.2.12 Specifying Endian Mode

To specify the endian mode, use the `be` arguments for the `-o` command-line option or the OPT directive:

`be`

Specifies big-endian object files: the most significant byte occupies the lower word address.

This first example uses the -o command-line option to specify big-endian mode:

```
scasm -b -l -obe vit.asm
```

This second example uses the OPT directive, placed at the beginning of the source file, to specify big-endian mode:

```
      opt  be
      page 132,42,0,0,0
LAB1  macro args
      ...
```

NOTE

For SC3900FP cores, the big-endian mode is enabled by default. Trying to specify the little-endian mode [the -ole command] generates error messages.

2.2.13 Checking Programming Rules

The reference manual for each core explains the rules for grouping and sequencing instructions in a variable length execution set (VLES). The assembler enforces static programming rules, marking violations at assembly time. Assembly does not take place if such errors exist. To keep the object file even if there is such an error, use the OPT svo (`-osvo`) option.

Rule identifiers begin actual error messages. For example, the identifier A.1 corresponds to Rule A.1 of the SC3900FP Core Reference Manual.

NOTE

The assembler's default setting for rule checking is ON, except for specific restriction checking. (This is equivalent to the `-s strict` option.)

Use these `-s` option patterns to control rule checking:

`-s id[,id...]`

Enables checking for violations of specified rules; `id` values are rule identifiers, without periods or other characters. Commas without spaces must separate multiple `id` arguments. Neither the `-s` option nor the `id` arguments are case sensitive.

`-sall`

Enables checking for all static rules.

`-snone`

Disables checking for all static rules.

`-sstrict`

Enables checking for static rules that could generate error messages; does not enable checking for static rules that could generate warnings.

This first example enables checking for all rules, in source file `myprog.asm`:

```
scasm
 -b -sall myprog.asm
```

This second example specifies checking for violations of the A.1, A.2, and G.G.1 rules, in source file `myprog.asm`:

```
scasm -b -sa1,a2,gg1 myprog.asm
```

NOTE

The system checks programming rules for sequential code; it does *not* check rules across changes of flow. For example, the assembler issues an error message for these code lines, which violate restriction T1:

```
cmp.eq.x #0,d0,p0:p1

if.p0 move.l r0,d1
```

But the assembler does *not* issue that error message in response to the same violation if the code includes a change of flow, as in:

```
[
      cmp.eq.x #0,d0,p0:p1
      if.p1 bra label1
 ]
label1:
      if.p1 ld.l (r0),d0
```

2.2.13.1 Code Examples

The below listing shows C code appropriate for -O0 compilation.

Listing 2-1. C Example for -O0 Compilation

```
volatile extern int bcr;
```

```
volatile extern int psdmr;

void initialize()

{

      bcr = 0x10000000; /*EBM = 0*/

    psdmr = 0x90000000; /*EAMUX = 1*/

}

void main()

{

      initialize(); /*SIU13 violation*/

}
```

The following listing shows C code appropriate for -O3 compilation.

Listing 2-2. C Example for -O3 Compilation

```
#define BR0 0x100
#define MAR 0x20
```

.c file:

```
void initialize
{

      int *br;

      br = (int *)BR0;

      *br = 0x280000; /*PS,DECC != 0*/

}

void initialize_read_from_UPM(int addr)

{

      int *mar;

      mar = (int *)MAR;

      *mar = addr;

}

void main()

{

      initialize_UPM();

      initialize_read_from_UPM(4); /*ADDRESS = 4*/ /*SIU 8 violation*/

}
```

The following listing shows the assembly code.

Listing 2-3. Assembly Language Code

```
tfra.l #$6260,r0 ;si2cmr
move.l #$1900,d0 ;rfsd = tfsd = 0, ce = fe = 1, dsc = 1

st.l d0,(r0) ;CPM36 violation
```

2.2.13.2 Data Analysis Terms

These terms apply to the assembler's data analyzer:

- Calling convention - convention the restriction checker uses for external calls. This convention specifies the registers and memory addresses that the called routine affects.
- Call tree - list of calling routines and called routines, from a source file, depicted in tree form.
- Constant propagation - algorithm that propagates constant resources over a control-flow graph.
- Control-flow graph (CFG) - rooted, directed graph that provides information about the flow of a routine.

2.2.13.3 Data Analysis Limitations

Keep in mind these limitations for -k restriction checking:

- The stack content is not available to the checker, nor can the checker know the initial values of the (O)SP registers.
- The checker treats PUSHN and POPN instructions the same way it treats regular PUSH/POP instructions.
- The checker always operates as if the processor is in the normal processing state: SP is NSP and OSP is ESP.
- Two code sections in the same asm file prevents restriction checking. (The assembler starts both such sections at p:$0, letting the linker choose the real addresses. This means that at least two instructions start at p:$0.)
- Peripherals must have the same endianness as the assembled file.
- The checker cannot distinguish a call to address p:$0 from a call to an external routine

2.2.13.4 Initialization File

The restriction checker relies on the initialization file for address definitions, label values, and calling conventions. You also can use this file to specify the addresses of such special registers as Brx and ORx (for memory controllers).

To include comments in the initialization file, write # as the first character of each comment line.

These directives may appear in the initialization file:

.address register_name mem_address

Assigns the specified hexadecimal address to the specified register; the register *must* be a memory mapped register. Neither argument is case sensitive. The mem_address value has the format p:xxxxxxxx.

Example: .address Br0 p:ff801801

.call-conv call_conv_id

[deleted]=[list_of_deleted_regs_and_mem_addresses]

Defines a calling convention. The call_conv_id name may be any ASCII string, except that the restriction checker ignores duplicate names.

The list in brackets consists of registers and hexadecimal memory intervals, separated by commas. Register names must have r.reg_ prefixes, such as r:reg_r0 or r:reg_n3. Memory intervals must follow the format p:100-300. To specify a single address, follow the format p:200 or p:200-200.

Example:

```
.call_conv 1
      deleted=[r:reg_d0,r:reg_sp,p:ffff]
      deleted=[r:reg_r1,r:reg_r2]
```

.funcName=call_conv_id

Specifies the calling convention for a function.

Example: .func_fibonacci=1

If the restriction checker finds a call to an external routine, the checker searches the initialization file's list of function calling conventions:

- If the list includes the function name, the checker applies the specified calling convention.
- If the list does not include the function name, the checker uses the default convention: the called function changes all the core registers without affecting memory addresses.

As the constant propagation algorithm needs an existing CFG, indirect change-of-flow instructions should have extra target specifications. The compiler may provide this information. But manually assembled files should pass this information to the checker in the form of a comment. This comment should be either on the line of the COF instruction or the last line of the packet that contains the COF instruction.

In this sample code, such a comment includes the symbols `l1` and `l2`, the possible targets of the jump instruction:

```
[
    move.l #4660,d0
    jsr r0
]                              ;Lint_info: targets: l1 l2
```

`.never_return_symbols` `list_of_symbols`

Tells assembler to *not* return to the current function, if a conditional or unconditional jump instruction hits any external symbol of the input list. Pertains to such instructions as `bra`, `break`, `cont`, `contd`, and `jmp`. (You can use this directive to make sure that the current basic block will not have any successors.)

Example: `.never_return_symbols _abort,___QCtxtRestore`

`.value` `label value`

Assigns the specified hexadecimal value to the label; the label may be a symbol known during the linking stage. The label is case sensitive.

Example: `.value StArT ffff`

2.3 Assembler Processing

This topic describes assembler processing for StarCore devices.

The StarCore assembler passes through code three times, performing these operations:

1. First Pass:

- Gathers instruction sequence and ordering information.
- As appropriate, rearranges instructions, generates error messages and warnings.

2. Second Pass:
   - Reads source program.
   - Builds symbol and macro tables.
3. Third Pass:
   - Referring to the Pass 2 tables, generates the object file.
   - Produces the source listing.

The assembler processes each source statement completely before reading the next statement. As it reads each line, the assembler applies all translations that DEFINE directives specify. Then the assembler examines the label, operation code, and operand fields. The assembler scans the macro definition table for matches with the operation code. If there is no match, the assembler scans the operation code and directive tables for matches with known opcode.

In case of an error, the assembler displays the appropriate error message, then the line that contains the error. The assembler displays all error messages, even if it does not generate a source listing. At the end of the source listing, the assembler prints error, warning, and remark totals. When the assembler returns control to the host operating system, it returns the number of errors as an exit status.

2.4 Source Statements

This topic describes the basic source statements for StarCore assembler.

Assembly language programs consist of two types of source statements:

- Assembly language instructions and a comment field
- An assembler directive and a comment field.

The StarCore assembly language supports conditional assembly. It also supports macros that replace a single program statement with the statements of the macro definition.

The following figure shows the four fields of the simplest source statement: label, operation, operand, and comment. Later sections of this chapter explain each field.

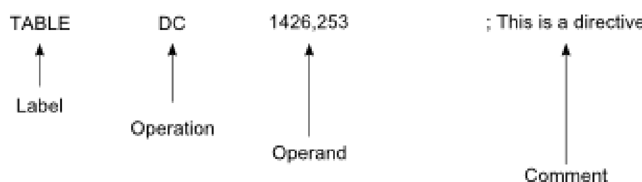


Figure 2-1. Basic Source Statement

Spaces or tabs must separate fields. The label, operation, and operand fields must not include spaces, except for spaces in quoted strings.

Only the first three fields are significant for the assembler; it ignores the comment field. The assembler treats anything beginning in column 1 as a label.

To extend a source statement to multiple lines, end all but the last line with the continuation character (\). Exception: An instruction group can span multiple lines without continuation characters, provided that brackets ([]) enclose the group.

Assembler mnemonics and directives are not case sensitive. But case does matter for labels, symbols, directive arguments, and literal strings.

If the source file contains horizontal tab characters (ASCII $09), the assembler moves them to the next fixed tab stop. The default stops are at eight-character intervals: columns 1, 9, 17, and so forth, but you can use the TAB directive to change the stops.

2.4.1 Label Field

Labels begin in column 1 of a source statement. If a line's first character is a space or tab, it probably means that the label field is empty. Label rules are:

- Label names must follow the same conventions as symbol names.
- A label whose first character is an underscore (_) is a *global label.*
- A label whose first character is a percent sign (%) is a *local label.*
- To indent a label, end it with a colon (:). Only space or tab characters may precede such an indented label.
- A label may occur only once in the label field of an individual source file, unless it is a local label or is used with the SET directive. If any non-local label occurs more than once in a label field, the assembler flags all references but the first as errors.
- A line may consist of only a label. Such a line assigns the value of the location counter to the label. Except for some directives, the assembler assigns a label the location-counter value for the first word of the instruction or data being assembled.

2.4.2 Operation Field

The operation field follows the label field; at least one space or tab must precede the operation field. Operation-field entries may be:

- Opcodes - Mnemonics that correspond directly to DSP machine instructions.

- Directives - Special assembler operation codes that control the assembly process.
- Macro calls - Invocations or macros, already defined.

The assembler first searches for operation codes in an internal macro definition table. If it does not find a match, it searches the table of machine operation codes and assembler directives. If neither of the tables holds the specified operation code, the assembler generates an error message.

To change this sequence, you can use the MACLIB directive. This means that macro names can replace standard machine operation codes and assembler directives, although the assembler issues warnings about such replacements.

2.4.3 Operand Field

The effect of the operand field depends on the contents of the operation field. Any operand-field value must follow the operation field; at least one space or tab must precede the operand value. Operand values may include symbols, expressions, or a combination of both; commas without spaces must separate multiple symbols or expressions.

As well as an operand value, the operand field includes the addressing mode for the instruction. For addressing mode definitions, see the core reference manual for your processor.

2.4.4 Comment Field

The assembler ignores comments, but you should include them in your source files for internal documentation. A comment field consists of a semicolon (;), followed by any characters that are not part of a literal string.

If a comment starts in the first column of the source file, the assembler aligns it with the label field. Otherwise, the assembler aligns comments the comment field. To prevent comments' reproduction in the source listing (or to prevent them being saved in macro definitions), start the comments with two successive semicolons (;;).

2.4.5 Variable Length Execution Sets

The StarCore architecture supports variable length execution sets (VLESes): grouping multiple instructions for parallel execution. For VLES grouping and sequencing rules, see the core reference manual for your processor.

The assembler interprets each line containing instructions as a VLES. Tabs or spaces must separate instructions, as this example shows:

```
ld.f (r2)+,d0  ld.f (r3)+,d8  subc.wo.leg.x d0,d0,d5 ;VLES, 3
instructions
```

To have a VLES span several lines, use bracket delimiters ([ ]), as this example shows:

```
[
  mac.leg.x d0.h,d1.h,d2  ; multiply operands
  add.x d0,d1,d3          ; add operands
  ld.f (r0)+,d0          ; load operands
  ld.w (r1)+,d1
]
```

NOTE

Lines of this example include only two instructions and one comment. This practice improves readability; but it is not required.

You should separate DALU and AGU instructions in a VLES: start with DALU instructions and end with AGU instructions.

2.4.6 Symbol Names

Follow these conventions for symbol names:

- Names consist of one or more characters.
- Names cannot begin with number characters 0-9. Otherwise, names can be any combination of alphanumeric characters (A-Z, a-z, 0-9) and the underscore character (_).
- Names and other identifiers containing a period (.) are reserved for the system.
- Names are case sensitive, but you can use the -oIC option to override the distinction between upper-case and lower-case letters.
- Names, regardless of case, must not duplicate the names of StarCore core registers, instructions, or pseudo-instructions. The assembler reserves these names.

This table shows examples of symbol names:

Type	Example
Valid names	loop_1; ENTRY; _alpha_BRAVO_charlie
Invalid names	1st_loop; loop&go; $value
Reserved names	dcl; nop; r0; section ; loopstart2

2.4.7 Symbol Labels

You may use symbols as labels. To make a label local, start it with the percent character (%C). This limits the label's scope to the area between any two non-local labels. The only source statements that can refer to or define such a local label are the statements between the source lines that contain the non-local labels. A local label is useful as the terminating address of a DO loop, or any such location that must have a unique label, but is not significant for documenting the source file.

In a macro, however, the scope of local labels is the entire macro expansion, without regard to non-local labels. Accordingly, all local labels within a macro must be unique. You can use such local labels freely within a macro definition, without regard to the number of macro expansions.

The assembler treats non-local labels within a macro expansion as normal labels. This means that such labels cannot occur more than once, unless you use them with the SET directive.

2.4.8 Strings

Literal ASCII strings can be operands for some assembler directives; they also have limited use in expressions. Such a string is one or more ASCII characters enclosed by single quotes ('). To specify an apostrophe within a literal string, use two consecutive apostrophe characters.

The alternate string delimiter is the double quote (") character. If you use double quotes to enclose a string, the assembler expands any DEFINE directive symbols contained in the string.

NOTE

Be careful about using the double-quote character inside macros, where this character is a dummy argument string

operator. You can use the macro concatenation operator to escape a double-quoted string.

The concatenation operator (++) tells the assembler to consider two strings to be one. For example, the strings `'ABC'++'DEF'` and `'ABCDEF'` are identical to the assembler.

Use brackets ([ ]) to have the assembler extract a substring. For example, if the assembler encounters the expression `['abcdefg',1,3]`, it uses the string value `'bcd'`. Substrings are valid wherever strings are, and you can nest substrings.

The assembler includes functions for determining the length of a string, and the position of one string within another.

2.5 Source Listing

This topic describes the source listing for StarCore assembler.

The source listing consists of the original source statements, formatted for easier reading, as well as other information the assembler generates. Most listing lines correspond directly to a source statement. Listing lines that do not correspond directly to source statements are page headings, error messages, expansions of macro calls, or expansions of directives such as DC.

According to its default setting, the assembler sends the source listing to the standard output. Options are:

- Sending the source listing to a printer, file, null device, or other such arbitrary destination. For this option, use the I/O redirection facilities of the host operating system.
- Sending the source listing to an argument file of the `-l` command-line option. If the `-l` option lacks an argument file, the assembler creates a source listing. To name this listing, the assembler adds the `.lst` extension to the name of the first source file in the command line.
- Inhibiting the source listing, by using the IL (inhibit listing) option.

Note that the `-b` and `-l` command-line options allow a hyphen as an argument: this directs the corresponding output to the standard output stream. But unpredictable results may occur if your settings send both the object file and the source listing to the same output stream.

The assembler always sends error messages to the standard output, regardless of option settings.

2.5.1 Source Listing Example

The following figure shows an example of the source listing. Text immediately after the figure explains areas of interest:


```
StarCore 100 Assembler Version 6.5.1 spec 0.63 01-05-02 15:31:59 vittbak.asm  Page 1

Viterbi Traceback

1                               opt   cex,mex,mu,svo   ; set assembler options
2                               page  132,42,0,0,0     ; set listing dimensions
4
5                      SIMSETUP  macro args
6    m                          org p:args
7    m                DEC_STOR dcb $00,$01,$02,$03,$04,$05,$06,$07
8    m                        dcb  $08,$09,$0a,$0b,$0c,$0d,$0e,$0f


17   m                         ds    100
18   m                 ENDM
19
25                    SIMSETUP       $150
26 +  P:00000150               org   p:$150
27 d+ P:00000150   00 DEC_STOR dcb  $00,$01,$02,$03,$04,$05,$06,$07
d    0             1
d    0             2
d    0             3
d    0             4
d    0             5
d    0             6
   d               07
28 d+  P:00000158 08          dcb  $08,$09,$0a,$0b,$0c,$0d,$0e,$0f
d    0             9
0                  A

51   P:00000212  6D        deceq  d0                 ; adjust value for loop count
6        4
**** 52 [vittbak.asm 40]: ERROR --- Duplicate destinations in paired instructions.
52   P:00000214  63       [ asrr #3,d0  tfr d0,d1  move.w #5,d0 ]
3                C
                 D0

 1  Errors
0 W arnings
```

Figure 2-2. Assembler Source Listing

Areas of interest are:

- Banner - The first line of each page. The banner consists of the assembler and version number, the date and time of assembly, the source file name, and the listing page number.
- Titles - Line 2 displays the title and line 3 displays the subtitle, provided that you have defined these titles. (Use the TITLE and STITLE directives.) If you have not defined titles, these lines are blank.
- Line number - The first field of source listing lines shows the line number.
- Indicator - The second field of source listing lines is the macro definition/expansion column. Possible values are:

Indicator	Meaning
`m`	Macro definition in progress. (The assembler does not assemble these lines, but retains them for macro expansion.)
`+`	Macro expansion in progress.
`d`	Data expansion occurring. (The `-oCEX` option requested this expansion.)
`i`	Line skipped due to an IF-THEN-ELSE directive sequence.

- Address - The third field of source listing lines contains the memory space value.
- Location Counter - The fourth field of source listing lines contains the location counter value.
- Statement - Fields 5 and beyond of source listing lines contain the source statement. This statement contains one or more instructions, depending on usage of instruction groups.
- Error Message - The listing shows an error message above the line that contains the error. The message consists of the source file name, the source line number, the severity level (remark, warning, error, or fatal), and the message text. The message may also include information about incorrect symbols or fields.
- Message Counts - The listing ends with counts of the assembler errors and warnings.

Chapter 3
Expressions

This section lists and describes the expressions their attributes, constants, operators and functions for StarCore assembler.

An expression represents a value that can be an operand of an assembler instruction or directive. Expressions consist of symbols, constants, operators, and parentheses. Expressions may contain:

- User-defined labels, with their integer or floating-point values
- Integers
- Floating-point numbers
- ASCII literal strings

In general, you may not use space or tab characters between the terms and operators of assembler expressions. Otherwise, expressions follow the rules of algebra and boolean arithmetic.

In this chapter:

- Absolute and Relative Expressions
- Expression Memory Space Attributes
- Internal Expression Representation
- Constants
- Operators
- Operator Precedence
- Functions

3.1 Absolute and Relative Expressions

This topic describes the absolute and relative expressions for StarCore assembler.

If the assembler operates in relative mode, all address expressions must follow these definitions:

- Absolute expression - An expression that consists only of absolute terms, or is the result of two relative terms with opposing signs.
- Relative expression - An expression that consists of a relative term by itself or a relative term in combination with absolute terms.

Only these types of expressions retain meaningful values after program relocation. For example, if your program pairs relative terms with opposing signs, the result is the difference between the two relative terms - an absolute value. But if code adds together two positive relative terms, the result is unpredictable - it depends on the terms' computed values at relocation time.

3.2 Expression Memory Space Attributes

As the assembler evaluates an expression, it uses the associated integer or floating-point value in place of each expression symbol.

Each symbol also includes a memory space attribute: P (program) or N (none).

The result of an expression always has an associated memory space attribute:

- Label, constant, and floating-point expressions associated with the SET directive always have the memory space attribute N.
- The unary logical negate operator, relational operators, logical operators, and some functions return values that have the memory space attribute 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 operand.
- The results of expressions involving operands with different memory space attributes have the memory space attribute P.

The assembler treats the memory space attribute as a type, as high-level languages use type for variables. Symbols that have the memory space attribute P should be addresses, so their maximum values should not exceed the maximum address value of the DSP inclusive. Only symbols that have the memory space attribute N can have values greater than the target processor's maximum address.

The memory space is implicitly P if you use an address as the operand of a LOOP, branch, or jump-type instruction.

Immediate addressing expressions can have any memory space attribute.

3.3 Internal Expression Representation

The assembler's internal representation of expression values depends on the target-processor word size.

The assembler supports word and double-word integer formats. Although the actual storage size of an expression value depends on the result's magnitude, the assembler can represent signed integers as long as 64 bits.

Internal floating-point representation depends almost entirely on the host environment, but the assembler's usual storage format for floating-point values is double precision. This format consists of 64 bits: 53 bits for the mantissa, 11 bits for the exponent, and an implied binary point.

3.4 Constants

Constants represent data values that do not vary during program execution.

3.4.1 Numeric Constants

The following table explains the possible numeric constants.

Table 3-1. Numeric Constants

Type	Description	Examples
Binary	Percent sign (%) followed by string of binary digits (0,1)	%11010
Hexadecimal	Dollar sign ($) or 0x, followed by string of hexadecimal digits (0-9, A-F, a-f)	$12FF0x12FF$12ff0x12ff
Decimal integer	String of decimal digits (0-9). Optional grave accent (`) can start the string.	12345
Decimal floating point	String of decimal digits that includes a decimal point or the letter E. The digits after the letter E are the exponent.	6E10.62.7e2

3.4.2 String Constants

The assembler converts expression string constants to right-aligned, concatenated sequences of ASCII bytes. Null strings have the value 0.

Examples are:

```
'ABCD'        ($41424344)

'''79'        ($00273739)

'A'           ($00000041)

''            ($00000000)  ¨ null string

'abcdef'      ($61626364)

'abc'++'de'   ($61626364)
```

The size limit for string expressions is the long-word size of the target processor. If a string exceeds this number of characters, the assembler truncates the value and prints a warning. This restriction also applies to string constants that involve the string concatenation operator, except for the DC and DCB directives.

3.5 Operators

Most assembler operators pertain to both floating-point and integer values.

The assembler follows these rules:

- If both operands are integers, the result is an integer value.
- If both operands are floating point values, the result is a floating-point value.
- If one operand is a floating-point value and the other operand is an integer value, the assembler converts the integer to a floating-point value, then applies the operator. The result is a floating-point value.

The table given below explains assembler operators, noting those restricted to integer operands. The main use of the relational and logical operands is with the IF conditional-assembly directive, although you can use these operands in any expression.

Table 3-2. Assembler Operators

Type	Operator	Description
Unary	+	Plus - Returns the positive value of its operand.
	-	Minus - Returns the negative value of its operand.
	~	One's Complement - Returns the one's complement of its integer operand; cannot be used with floating-point operands.
	!	Logical Negate - Returns an integer 1 if the value of its operand is 0; otherwise returns a 0. The result's memory space attribute is N Example: If symbol `BUF` has the value 0, `!BUF` has the value 1. If `BUF` has the value 1000, `!BUF` has the value 0.
Arithmetic	+	Addition - Yields the sum of its operands.
	-	Subtraction - Yields the difference between its operands.
	*	Multiplication - Yields the product of its operands.
	/	Division - Yields the quotient: the first operand divided by the second. For integer operands, the result is a truncated integer.
	%	Mod - Yields the remainder of the first operand divided by the second. (If both operands are floating-point values and the divisor is 0.0, the result is the dividend.)
Shift	<<	Shift Left - For integer operands only. Shifts and zero fills the left operand to the left; the right operand specifies the number of bits to shift.
	>>	Shift Right - For integer operands only. Shifts the left operand to the right; the right operand specifies the number of bits to shift. Extends the sign bit.
Relational	<	Less Than - Returns integer 1 if the expression is true, integer 0 if the expression is false. The result's memory space attribute is N. Example: If D = 3 and E = 5, D < E = 1.

Table continues on the next page...

Table 3-2. Assembler Operators (continued)

<table>
<tr><th>Type</th><th>Operator</th><th>Description</th></tr>
<tr><td></td><td>></td><td>Greater Than - Returns integer 1 if the expression is true, integer 0 if the expression is false. The result's memory space attribute is N. Example: If D = 3 and E = 5, D > E = 0.</td></tr>
<tr><td></td><td><=</td><td>Less Than or Equal - Returns integer 1 if the expression is true, integer 0 if the expression is false. The result's memory space attribute is N.</td></tr>
<tr><td></td><td>>=</td><td>Greater Than or Equal - Returns integer 1 if the expression is true, integer 0 if the expression is false. The result's memory space attribute is N.</td></tr>
<tr><td></td><td>==</td><td>Equal - Returns integer 1 if the expression is true, integer 0 if the expression is false. The result's memory space attribute is N.</td></tr>
<tr><td></td><td>!=</td><td>Not Equal - Returns integer 1 if the expression is false, integer 0 if the expression is true. The result's memory space attribute is N.</td></tr>
<tr><td>Bitwise</td><td>&</td><td>AND - For integers only. Yields the bitwise AND function of its operands.</td></tr>
<tr><td></td><td>|</td><td>OR - For integers only. Yields the bitwise OR function of its operands.</td></tr>
<tr><td></td><td>^</td><td>Exclusive OR - For integers only. Yields the bitwise exclusive OR function of its operands.</td></tr>
<tr><td>Logical</td><td>&&</td><td>Logical AND - Returns integer 1 if both operands are nonzero; otherwise returns integer 0.</td></tr>
<tr><td></td><td>||</td><td>Logical OR - Returns integer 1 if either operand is nonzero; otherwise returns integer 0.</td></tr>
</table>

3.6 Operator Precedence

The assembler evaluates expressions from left to right.

Below listed are the rules of operator precedence the assembler follows:

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, exclusive OR
9. Logical AND, OR

Valid operands include numeric constants, literal ASCII strings, and symbols.

You cannot apply the one's complement, shift, or bitwise operators to floating-point operands. That is, if an expression evaluation results in a floating-point value on either side of any such operator, the assembler generates an error message.

3.7 Functions

The assembler's built-in functions support data conversion, string comparison, and transcendental math computations.

You may use functions as terms in any arbitrary expression; functions may have no arguments, one argument, or multiple arguments. These rules apply:

- Open and close parentheses must always follow functions.
- Arguments that are expressions must be absolute expressions, except where noted.
- Arguments must not contain external references.
- There must not be intervening spaces between the function name and the open parenthesis, or between comma-separated arguments.

The table given below lists the assembler functions of each type:

Table 3-3. Assembler Function List

Type	Function
Mathematical	ABS - Absolute value
	ACS - Arc cosine
	ASN - Arc sine
	AT2 - Arc tangent
	ATN - Arc tangent
	CEL - Ceiling
	COH - Hyperbolic cosine
	COS - Cosine
	FLR - Floor
	L10 - Log base 10

Table continues on the next page...

Table 3-3. Assembler Function List (continued)

Type	Function
	LOG - Natural logarithm
	MAX - Maximum value
	MIN - Minimum value
	POW - Raise to power
	RND - Random value
	SGN - Return sign
	SIN - Sine
	SNH - Hyperbolic sine
	SQT - Square root
	TAN - Tangent
	TNH - Hyperbolic tangent
	XPN - Exponential function
Conversion	CVF - Convert Integer to floating point
	CVI - Convert floating point to integer
	CVS - Convert Memory space
	FLD - Shift and mask
	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
String	LEN - String length
	POS - Position of substring
	SCP - Compare strings
Macro	ARG - Macro argument
	CNT - Macro argument count
	MAC - Macro definition
	MXP - Macro expansion
Assembler Mode	BIGENDIAN - Endian mode check
	CCC - Cumulative cycle count
	CHK - Instruction/data checksum
	CTR - Location counter number
	DEF - Defined symbol
	EXP - Expression check
	INT - Integer check
	LCV - Location counter value
	LST - LIST directive flag value
	MSP - Memory space

Table continues on the next page...

Table 3-3. Assembler Function List (continued)

Type	Function
	REL - Relative mode

Descriptions of the assembler functions complete this chapter. These descriptions are in alphabetic order, without regard to function types. Although these descriptions show functions in upper case, the functions are not case sensitive.

3.7.1 ABS Absolute Value

Returns the absolute value of the specified expression, as a floating-point value. The result's memory space attribute is N.

```
@ABS(expr)
```

Parameter

`expr`

Any valid expression.

Example:

```
MOVE.L #@ABS(VAL),D4       ; Load absolute value
```

3.7.2 ACS Arc Cosine

Returns the arc cosine of the specified expression, as a floating-point value, in the range zero to pi. The result's memory space attribute is N.

```
@ACS(expr)
```

Parameter

`expr`

Any valid expression that evaluates to a value between -1 and 1.

Example:

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

3.7.3 ARG Macro Argument

Returns integer 1 if the specified macro argument is present; otherwise returns 0. The result's memory space attribute is N.

```
@ARG(symbol | expr)
```

Parameters

`symbol`

Any valid symbol that refers to a dummy argument name; must be in quotes.

`expr`

Any valid expression that refers to the argument's ordinal position in the macro dummy argument list.

Remarks

If you use this function when no macro expansion is active, the assembler issues a warning.

Example:

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

3.7.4 ASN Arc Sine

Returns the arc sine of the specified expression, as a floating-point value, in the range -pi/2 to pi/2. The result's memory space attribute is N.

```
@ASN(expr)
```

Parameter

`expr`

Any valid expression that evaluates to a value between -1 and 1.

Example:

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

3.7.5 AT2 Arc Tangent

Returns the arc tangent of the quotient of two expressions (`expr1/expr2`), as a floating-point value, in the range -pi to pi. A comma must separate the `expr1` and `expr2` expressions. The result's memory space attribute is N.

```
@AT2(expr1,expr2)
```

Parameters

`expr1, expr2`

Any valid expressions.

Example:

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

3.7.6 ATN Arc Tangent

Returns the arc tangent of the specified expression, as a floating-point value, in the range -pi/2 to pi/2. The result's memory space attribute is N.

```
@ATN(expr)
```

Parameter

`expr`

Any valid expression.

Example:

```
MOVE.L #@ATN(1.0),D0
 ; Load arc tangent
```

3.7.7 BIGENDIAN Endian Mode Check

Returns an integer 1, as big-endian mode is always enabled for SC3900FP.

```
@BIGENDIAN()
```

Example:

```
   IF @BIGENDIAN()
   DCB "BIG-ENDIAN"
ELSE
   DCB "LITTLE-ENDIAN"
ENDIF
```

NOTE

Another way to check for big-endian compiling of an asm file is to use @DEF:

```
IF @DEF(`__BIG_ENDIAN__')
 DCB 1
ELSE
 DCB 0
ENDIF
```

3.7.8 CCC Cumulative Cycle Count

Returns the cumulative cycle count as an integer; useful with the CC, NOCC, and CONTC assembler options. The result's memory space attribute is N.

```
@CCC()
```

Example:

```
IF @CCC() > 200 ; Check if cycle count > 200
```

3.7.9 CEL Ceiling

Returns the ceiling of the specified expression: a floating-point value that represents the smallest integer greater than or equal to the expression. The result's memory space attribute is N.

```
@CEL(expr)
```

Parameter

`expr`

Any valid expression.

Example:

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

3.7.10 CHK Instruction/Data Checksum

Returns the current instruction/data checksum value as an integer. The result's memory space attribute is N.

```
@CHK()
```

Remarks

Useful in conjunction with the CK, NOCK, and CONTCK assembler options . Note that using directives other than SET to assign the checksum value could lead to phasing errors, due to different generated instruction values between passes.

Example:

```
CHKSUM SET @CHK() ; Reserve checksum value
```

3.7.11 CNT Macro Argument Count

Returns the count of the current macro expansion arguments as an integer. If you use this function when no macro expansion is active, the assembler issues a warning. The result's memory space attribute is N.

```
@CNT()
```

Example:

```
ARGCNT SET @CNT() ; Reserve arg count
```

3.7.12 COH Hyperbolic Cosine

Returns the hyperbolic cosine of the specified expression, as a floating-point value. The result's memory space attribute is N.

```
@COH(expr)
```

Parameter

`expr`

Any valid expression.

Example:

```
HYCOS EQU @COH(VAL) ; Compute hyperbolic cosine
```

3.7.13 COS Cosine

Returns the cosine of the specified expression, as a floating-point value. The result's memory space attribute is N.

```
@COS(expr)
```

Parameter

`expr`

Any valid expression.

Example:

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

3.7.14 CTR Location Counter Number

Returns the counter number of the specified location counter. The returned counter number is an integer value with memory space attribute N.

```
@CTR(L|R)
```

Parameters

`L`

Specifier for the load location counter.

`R`

Specifier for the runtime location counter.

Example:

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

3.7.15 CVF Convert Integer to Floating Point

Converts the value of the specified expression to a floating-point value. The result's memory space attribute is N.

```
@CVF(expr)
```

Parameter

`expr`

Any valid integer expression.

Example:

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

3.7.16 CVI Convert Floating Point to Integer

Converts the value of the specified expression to an integer value. The result's memory space attribute is N. (Such conversions can be inexact, possibly truncating floating-point values.)

```
@CVI(expr)
```

Parameter

`expr`

Any valid floating-point expression.

Example:

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

3.7.17 CVS Convert Memory Space

Assigns the specified memory space attribute to the specified expression, returning the same expression.

```
@CVS(P | N,expr)
```

Parameters

`P`

Specifier for memory space attribute P.

`N`

Specifier for memory space attribute N.

`expr`

Any valid relative or absolute expression.

Example:

```
LOADDR EQU @CVS(P,TARGET) ; Set LOADDR to P:TARGET
```

3.7.18 DEF Defined Symbol

Returns an integer 1 if the specified symbol is defined; otherwise returns a 0. The result's memory space attribute is N.

```
@DEF(symbol)
```

Parameter

`symbol`

Any label not associated with a MACRO or SECTION directive. Quotes tell the assembler to look for a DEFINE symbol; if `symbol` is not in quotes, the assembler looks for an ordinary label.

Example:

```
IF @DEF(ANGLE) ; Assemble if ANGLE is defined
```

3.7.19 EXP Expression Check

Returns an integer 1 if evaluating the specified expression would result in errors; otherwise returns 0. The result's memory space attribute is N. The assembler does not issue an error message if the expression contains an error; the assembler does not test for warnings.

```
@EXP(expr)
```

Parameter

`expr`

Any valid relative or absolute expression.

Example:

```
IF !@EXP(@FRC(VAL)) ; Skip on error
```

3.7.20 FLD Shift and Mask

Shifts and masks the value expression into the base expression for width bits, beginning at the start bit. If you omit the start-bit value, the assembler uses zero (the least significant bit). Returns the shifted and masked value, with memory space attribute N.

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

Parameters

`base`

Original positive-integer expression; may not exceed the target word size.

`value`

Positive-integer expression shifted and masked into `base`; may not exceed the target word size.

`width`

Number of bits to shift; a positive-integer expression that may not exceed the target word size.

`start`

Optional: Starting bit for the operation; a positive-integer expression that may not exceed the target word size. .

Example:

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

3.7.21 FLR Floor

Returns the floor of the specified expression: a floating-point value that represents the largest integer less than or equal to the expression. The result's memory space attribute is N.

```
@FLR(expr)
```

Parameter

`expr`

Any valid expression.

Example:

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

3.7.22 FRC Convert Floating Point to Fractional

Scales and convergent rounds a floating-point expression, returning its fractional representation as an integer. The result's memory space attribute is N.

```
@FRC(expr)
```

Parameter

`expr`

Any valid floating-point expression.

Example:

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

3.7.23 INT Integer Check

Returns an integer 1 if the specified expression evaluates to an integer; otherwise returns a 0. The result's memory space attribute is N.

```
@INT(expr)
```

Parameter

`expr`

Any valid relative or absolute expression.

Example:

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

3.7.24 L10 Log Base 10

Returns the base 10 logarithm of the specified expression, as a floating-point value. The result's memory space attribute is N.

```
@L10(expr)
```

Parameter

`expr`

A numerical expression greater than zero.

Example:

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

3.7.25 LCV Location Counter Value

Returns the memory space attribute and value of the specified location counter. The optional second argument indicates the Low, High, or numbered counter; a comma must separate the two arguments. If you omit the second argument, the assembler uses the default counter (counter 0).

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

Parameters

`L`

If the first argument, specifier for the load location counter.

If the optional second argument, specifier for the low counter.

`R`

Specifier for the runtime location counter.

`H`

Specifier for the high counter.

`expr`

Specifier for a numbered counter; must evaluate to an integer value.

Remarks

This function does not work correctly if you use it to specify the runtime counter value of a relocatable overlay. This is because the resulting value is an overlay expression, and you may not use overlay expressions to set the runtime counter for a subsequent overlay.

Example:

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

3.7.26 LEN String Length

Returns the length of the specified string, as an integer. The result's memory space attribute is N.

```
@LEN(string)
```

Parameter

`string`

Any valid string.

Example:

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

3.7.27 LFR Convert Floating Point to Long Fractional

Scales and convergent rounds a floating-point expression, returning its fractional representation as a long integer. The result's memory space attribute is N.

```
@LFR(expr)
```

Parameter

`expr`

Any valid, floating-point expression.

Example:

```
LFRAC EQU @LFR(LFLT) ; Store binary form
```

3.7.28 LNG Concatenate to Double Word

Concatenates single words into a double word: `expr1` becomes the high word, `expr2` becomes the low word. The result's memory space attribute is N.

```
@LNG(expr1,expr2)
```

Parameters

`expr1, expr2`

Any valid, single-word expressions.

Example:

```
LWORD DC @LNG(HI,LO) ; Build long word
```

3.7.29 LOG Natural Logarithm

Returns the natural logarithm of the specified expression, as a floating-point value. The result's memory space attribute is N.

```
@LOG(expr)
```

Parameter

`expr`

Any valid expression greater than zero.

Example:

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

3.7.30 LST LIST Directive Flag Value

Returns the value of the LIST directive flag as an integer, with memory space attribute N. (Each time the assembler encounters the LIST directive in source code, it increments the flag; each time it encounters the NOLIST directive, it decrements the flag.)

```
@LST()
```

Example:

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

3.7.31 LUN Convert Long Fractional to Floating Point

Converts a double-word long fractional to a floating-point value. The result's memory space attribute is N.

```
@LUN(expr)
```

Parameter

`expr`

A binary fraction expression.

Example:

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

3.7.32 MAC Macro Definition

Returns integer 1 if the specified symbol is defined as a macro name; otherwise returns 0. The result's memory space attribute is N.

```
@MAC(symbol)
```

Parameter

`symbol`

Any valid symbol.

Example:

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

3.7.33 MAX Maximum Value

Determines which input expression has the greatest value, then returns that expression as a floating-point value. The result's memory space attribute is N.

```
@MAX(expr1[,...,exprN])
```

Parameters

`expr1 ... exprN`

Any valid expressions.

Example:

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

3.7.34 MIN Minimum Value

Determines which input expression has the least value, then returns that expression as a floating-point value. The result's memory space attribute is N.

```
@MIN(expr1[,...,exprN])
```

Parameters

`expr1 ... exprN`

Any valid expressions.

Example:

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

3.7.35 MSP Memory Space

Returns the memory space attribute of the specified expression, as integer value 0 (for N) or 4 (for P).

```
@MSP(expr)
```

Parameter

`expr`

Any valid relative or absolute expression.

Example:

```
MEM SET @MSP(ORIGIN) ; Save memory space
```

3.7.36 MXP Macro Expansion

Returns an integer 1 if the assembler is expanding a macro; otherwise returns a 0. The result's memory space attribute is N.

```
@MXP()
```

Example:

```
IF @MXP() ; Macro expansion active?
```

3.7.37 POS Position of Substring

Returns the position of substring str2 in source string str1 as an integer. Begins search at position start. If you omit the start value, the search begins at the beginning of str1. The result's memory space attribute is N.

```
@POS(str1,str2[,start])
```

Parameters

`str1`

Source string.

`str2`

Substring; must not exceed the length of `str1`.

`start`

A positive integer expression that does not exceed the length of `str1`.

Example:

```
ID EQU @POS('Star*Core 140','Core') ; ID = 5
```

3.7.38 POW Raise to a Power

Returns the first expression, raised to the power of the second expression, as a floating-point value. A comma must separate the two expressions. The result's memory space attribute is N.

```
@POW(expr1,expr2)
```

Parameters

`expr1`

The expression whose value is to be raised.

`expr2`

The power-value expression.

Example:

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

3.7.39 REL Relative Mode

Returns an integer 1 if the assembler is operating in relative mode; otherwise returns a 0. The result's memory space attribute is N.

```
@REL()
```

Example:

```
IF @REL() ; Check if in relative mode
```

3.7.40 RND Random Value

Returns a random value in the range 0.0 to 1.0. The result's memory space attribute is N.

```
@RND()
```

Example:

```
SEED DC @RND() ; Save initial seed value
```

3.7.41 RVB Reverse Bits in Field

Reverses bits of the first expression, in the field the second expression delimits. Omitting the second expression makes the bit-reverse field the target word size.

```
@RVB(expr1[,expr2])
```

Parameters

`expr1, expr2`

Single-word, integer expressions.

Example:

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

3.7.42 SCP Compare Strings

Returns an integer 1 if the specified strings are the same; otherwise returns 0. A comma must separate the two strings. The result's memory space attribute is N.

```
@SCP(str1,str2)
```

Parameters

`str1, str2`

String expressions.

Example:

```
IF @SCP(STR,'MAIN') ; Check if STR equals MAIN
```

3.7.43 SGN Return Sign

Returns the sign of the specified expression as an integer: -1 (negative), 0 (zero), or 1 (positive). The result's memory space attribute is N.

```
@SGN(expr)
```

Parameter

`expr`

Any valid relative or absolute expression.

Example:

```
IF @SGN(INPUT) ; Check if sign is positive
```

3.7.44 SIN Sine

Returns the sine of the specified expression, as a floating-point value. The result's memory space attribute is N.

```
@SIN(expr)
```

Parameter

`expr`

Any valid expression.

Example:

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

3.7.45 SNH Hyperbolic Sine

Returns the hyperbolic sine of the specified expression, as a floating-point value. The result's memory space attribute is N.

```
@SNH(expr)
```

Parameter

`expr`

Any valid expression.

Example:

```
HSINE EQU @SNH(VAL) ; Hyperbolic sine
```

3.7.46 SQT Square Root

Returns the square root of the specified expression, as a floating-point value. The result's memory space attribute is N.

```
@SQT(expr)
```

Parameter

`expr`

Any valid positive expression.

Example:

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

3.7.47 TAN Tangent

Returns the tangent of the specified expression, as a floating-point value. The result's memory space attribute is N.

```
@TAN(expr)
```

Parameter

`expr`

Any valid expression.

Example:

```
MOVE.L #@TAN(1.0),D1
; Load tangent
```

3.7.48 TNH Hyperbolic Tangent

Returns the hyperbolic tangent of the specified expression, as a floating-point value. The result's memory space attribute is N.

```
@TNH(expr)
```

Parameter

`expr`

Any valid expression.

Example:

```
HTAN = @TNH(VAL) ; Hyperbolic tangent
```

3.7.49 UNF Convert Fractional to Floating Point

Converts a fractional to a floating-point value. The result's memory space attribute is N.

```
@UNF(expr)
```

Parameter

`expr`

A binary fraction expression.

Example:

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

3.7.50 XPN Exponential Function

Returns the exponential function (base e raised to the power of the specified expression), as a floating-point value. The result's memory space attribute is N.

```
@XPN(expr)
```

Parameter

`expr`

Any valid expression.

Example:

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

Chapter 4
Software Project Management

Complex software projects often consist smaller program units.

A team of programmers may write these subprograms in parallel, or they may reuse subprograms of a previous development effort.

This chapter explains the assembler directives that help manage complex software projects.

In this chapter:

- Sections
- Sections and Relocation
- Address Assignment
- Overlays
- Multi-Programmer Environment Example

4.1 Sections

The SECTION and ENDSEC directives encapsulate program units.

This defines relocatable blocks of code and data, postponing concerns about memory placement until after the assembly process.

A SECTION directive defines the start of a section, giving it the name that `section_name` specifies. The ENDSEC directive specifies the end of the section. The format is:

```
SECTION section_name [GLOBAL |STATIC|LOCAL] [core_id`]
.
.
source statements
.
.
ENDSEC
```

NOTE

Except for debug sections, text sections have the default alignment of 2.

4.1.1 Section Names

Although you may give almost any name to a section, the assembler recognizes the names of conventional ELF sections - .text, .data, .rodata, and .bss. The below table lists the default types and flags for these section names. The assembler treats sections with other names as code (.text) sections, setting types and flags accordingly. If such a section is not a code section, you must use the SECTYPE and SECFLAGS directives to override the default settings.

NOTE

Multiple sections can have the same name, provided that they also have the same type. Otherwise, the assembler issues an error message.

Table 4-1. Conventional ELF Sections

Section	Contents	Type	Attributes
.bss	Uninitialized data	NOBITS	ALLOC, Write
.data	Initialized data	PROGBITS	ALLOC, WRITE
.mw_info	Assembler-generated contents that the linker consumes during dead data stripping	SHT_MW_INFO (SHT_LOPROC+3)	no sh_flags(0)
.note	User comments, as ABI 2.0 defines.	SHT_NOTE(7)	no sh_flags(0)
.rodata	Read-only, initialized data	PROGBITS	ALLOC
.text	Program code	PROGBITS	ALLOC, EXECINSTR

The table given below lists the reserved names for specialized ELF sections; you should not use any of these names.

Table 4-2. Reserved Section Names

.debug_abbrev	.debug_pubname	.rel_line
.debug_aranges	.default	.rel.line.debug_info
.debug_info	.line	.shstrtab
.debug_line	.mw_info	.strtab
.debug_loc	.note	.symtab
.debug_macro	.rel.debug_loc	

4.1.2 Nested and Fragmented Sections

You can nest sections to any level. When the assembler encounters a nested section, it stacks the current section and uses the new (child) section. When the assembler reaches the ENDSEC directive of the nested section, the assembler restores and resumes using the parent section. The ENDSEC directive always pertains to the most recent SECTION directive.You also can split sections into separate parts by using the same section name with multiple SECTION and ENDSEC directive pairs. Reusing a section name lets you arrange source statements arbitrarily, for example, grouping all statements that reserve P space storage locations.

4.1.3 Sections and Symbols

The default arrangement is that symbols defined within a section are local symbols. Any reference to a local symbol can be satisfied in the file in which it is defined.

Defining symbols outside a section makes them global. Such symbols can satisfy an outstanding current-file reference at assembly time, or an outstanding reference in any file at link time. Code inside or outside any section may reference global symbols freely, as long as the global symbol does not conflict with another symbol of the same name.

To declare a section's local symbols global:

- Use the GLOBAL directive for an individual symbol.
- Use the GLOBAL qualifier of the SECTION directive for all symbols in a section.

In the listing given below `SYM1` and `SYM2` are global symbols, initially defined outside any section. But section `EXAMPLE` defines `SYM1` locally, with a different value:

- This interior redefinition means that the first `MOVE` instruction moves the value 3 to R0.
- `SYM2` remains a global symbol, so the second `MOVE` instruction moves the value 2 to R1.
- The final `MOVE` instruction is outside any section, so it uses the global `SYM1` definition, moving the value 1 to R2.

Listing 4-1. Sections and Data Hiding

```
SYM1     EQU 1
SYM2     EQU 2
```

```
        SECTION EXAMPLE
SYM1    EQU 3
        TFRA.L #SYM1,R0
        TFRA.L #SYM2,R1
        ENDSEC
        TFRA.L #SYM1,R2        MOVE #SYM1,R2
```

4.1.4 Macros and DEFINE Symbols within Sections

Macros and DEFINE directive symbols you define within a section are local. Global access never is possible for such macros and symbols.

To make macros or DEFINE symbols accessible globally, you must define them outside any section.

4.2 Sections and Relocation

Sections are the basic groups for relocating code and data blocks.

With respect to relocation, code or data inside a section is an indivisible block, bound to a memory space. Within this memory space; such a code or data block is independently relocatable.

The assembler allocates a set of P-memory-space location counters for each section the source code defines. The assembler uses these counters to maintain data and instruction offsets from the beginning of the section. At link time, the linker can relocate sections to absolute addresses, load them in a particular order, or link them contiguously, as the programer specifies. If sections are split into parts or dispersed among files, the linker can recombine them logically, permitting relocation as a unit for each section.

Sections may be relocatable or absolute. If the assembler runs in absolute mode (command line -a option), all sections are absolute. If the assembler runs in relative mode (the default), all sections are relocatable initially.

To make a section or a part of a section absolute, use an ORG directive that specifies an absolute runtime address. If the assembler encounters such an ORG directive, it switches to absolute mode, generating absolute addresses. The assembler continues generating absolute code until it encounters an ENDSEC directive.

4.3 Address Assignment

This topic describes the address assignment supported by StarCore Assembler.

The StarCore assembler supports either:

- Assigning absolute addresses at assembly time
- Generating relocatable program addresses to be resolved during linking

The assembler allows two sets of program counters: load counters and runtime counters. This distinction lets the assembler support *overlays* - runtime code/data transfers from one address to another. For example, you might load code and data at addresses A and B, but overlays could copy them to addresses F and N for execution.

Use the ORG directive to specify absolute address assignment. This directive also can specify the location counter (H, L, default, or a section's numbered counter), and assign initial values.

NOTE

Counters 0, 1, and 2 correspond to the default, L, and H counters. Apart from this, there is no inherent relationship among numbered counters.

Location counter names default, L, and H are symbolic - the assembler does not verify that an H-counter value is greater than an L-counter value.

Counters are useful for providing mnemonic links among individual memory blocks. You can use separate counters for blocks in one section that get mapped to separate physical memories.

4.4 Overlays

When you use the SECTION directive, you define a regular section; you may use the SECFLAGS or SECTYPE directives to modify the section.

To define an overlay section, use the OVERLAY operand of the SECTYPE directive. Use the SECTYPE UNION directive to define a data overlay, as this example shows:

```
section .ovl_star local                  section.ovl_pure_data
secflags nowrite,alloc,execinstr         secflags alloc
sectype overlay                              sectype union
```

Each overlay section has two starting addresses:

- Load address - address where the linker links the section.
- Run address - address where the section begins during execution of its code.

All symbol references symbols in an overlay section refer to the run address. To refer to a global symbol's load address, prefix the name with `LoadAddr_`, as shown in the below listing. (You cannot make local symbols refer to load addresses.)

Listing 4-2. Referencing a Global Symbol's Load Address

```
      section .text local
      global _main

_main:

      push.l r0

      tfra.l #LoadAddr__star,r0

      jsr __overlay_manager

      . . .

      pop.l r0

      jsr _star

      rts

      section .ovl_text local

      secflags alloc,execinstr,nowrite

      sectype overlay

      global _star

      _star:

      . . .

      rts
```

An overlay manager must copy an overlay section from its load address to its run address. In the above listing, the code calls the overlay manager to load the overlay section `.ovl_text` to its run address. The overlay manager must know the overlay section's load address, which is unique (many overlays could run at the same address). After the overlay manager finishes, it is safe to call code in the overlay section.

You must provide a symbol at the start of each overlay section; the assembler does not create these symbols automatically.

4.4.1 Overlay Manager

The listing provided below shows a basic C implementation of an overlay manager:

Listing 4-3. Basic Overlay Manager

```
#include <stdlib.h>
#include <stdio.h>

#include <string.h>

struct ovltab

 {

  void *ovl_run;

  void *ovl_load;

  unsigned long int ovl_size;

  unsigned long int ovl_checksum;

  unsigned long int ovl_flags;

  unsigned long int ovl_other;

  unsigned short int ovl_parent;

  unsigned short int ovl_sibling;

  unsigned short int ovl_child;

 }

extern struct ovltab _overlay_table[];

extern unsigned long int _overlay_count;

void *

_overlay_manager(void *load_addr)

 {

  unsigned long int i;

  for(i=0;i<_overlay_count; ++i){

      if(_overlay_table[i].ovl_load == load_addr){

        return memcpy(_overlay_table[i].ovl_run,

            _overlay_table[i].ovl_load,

            _overlay_table[i].ovl_size);

      }

  }

 return NULL;
```

```
}
```

4.4.2 Overlay Example

The below listing shows a sample assembly source file, `ovl.asm`, that defines overlay sections `ovl_star1` and `ovl_comet1`. To assemble this file, use the command:

```
scasm -l -b ovl.asm
```

Listing 4-4. ovl.asm Assembly Source File

```
; main calling star1 and comet1
      section .text local

      global _main

_main:

      push.l r0

      tfra.l #LoadAddr__star1,r0

      jsr __overlay_manager

      pop.l r0

      jsr _star1

      push.l r0

      tfra.l #LoadAddr__comet1,r0

      jsr __overlay_manager

      pop.l r0

      jsr _comet1

      rts

      endsec

      section .ovl_star1 local

      secflags alloc,execinsr,nowrite

      sectype overlay

      global _star1

_star1:

      subc.wo.x d0,d0,d1

      addc.wo.x #5,d1,d1

      rts

      endsec
```

```
        section .ovl_comet1 local
        secflags alloc,execinsr,nowrite
        sectype overlay
        global _comet1
_comet1:
        subc.wo.x d0,d0,d2
        addc.wo.x #5,d2,d2
        rts
        endsec
```

The following command links the resulting object file, using the linker command file shown in the listing below.

```
  sc3000-ld -arch b4860 -Map test.map -c test.l3k test.eln
```

NOTE

For MMU specific architecture, b4860 target architecture must be used.

Listing 4-5. ovl.cmd Linker Command File

```
;*******************************************************************
;*
;* This memory control file defines the memory layout used with
;* the sc3900fp simulator. This file assumes small memory model
;* (i.e all globals and static data fit in lower 64k)
;*
;*    0          - 0x1ff             Interrupt vectors and handlers
;*  DataStart    - DataStart+DataSize-1      Global and static data
;*  CodeStart    - StackStart-1     Application code
;*  StackStart   - TopOfStack       stack/heap
;*  ROMStart     - TopOfMemory      ROM
;*
;*******************************************************************
;
; Define configuration specific values
;
.provide _DataStart,   0x0200        ; Start of global and static data
.provide _DataSize,  0x10000
```

```
.provide _CodeStart, 0x100000        ; Sets the code start address

.provide _StackStart,0x200000        ; Sets the stack start address

                                     ; The stack grows upwards.

.provide _TopOfStack, 0x2fff00       ; The highest address to be used

                                     ; by the C/C++ run-time.

                                     ; By default, this serves as the

                                     ; heap start address.

                                     ; The heap grows downwards.

.provide _ROMStart,    0x300000      ; Sets the ROM start address

.provide _TopOfMemory, 0x3fffff      ; The highest address in memory

.provide _SR_Setting, 0xe4000c       ; Value to set the SR after
reset:

                                     ;  exception mode

                                     ;  interrupt level 7

                                     ;  saturation on

                                     ;  rounding mode: nearest even

.memory 0, _TopOfMemory, "rwx"       ; Start execution at interrupt

                                     ; vector first entry (RESET).

.reserve _DataStart+_DataSize-2, _DataStart+_DataSize-1

                                     ; Reserve the top of the data
area

                                     ; to generate a linker error if

                                     ; data size does not fit 64k.

.reserve _StackStart, _TopOfStack    ; Reserve for stack and heap
space

.entry 0                             ; IntVec

.org 0

.overlay ".overlay1", "rwx", ".ovl_star1", ".ovl_star2", ".ovl_star3"

.overlay ".overlay2", "rwx", ".ovl_comet1", ".ovl_comet2"

.segment .intvec, ".intvec"

.org _DataStart

.segment .data,".data",".ramsp_0",".default",".bss"

.segment .ovltab, ".ovltab"

.org _CodeStart

.segment .text, ".text"

.segment .ovlstar, ".ovl_star*"
```

```
.segment .ovlcomet, ".ovl_comet*"

.segment .overlay1, ".overlay1"

.segment .overlay2, ".overlay2"

.org _ROMStart

.segment .rotable, ".init_table"

.segment .roinit,

 ".rom_init"
```

4.5 Multi-Programmer Environment Example

Developers often split multi-programmer projects into tasks that represent functional units.

For example, suppose that a project has three task divisions: i/o, filter, and main. An individual programmer writes each task as a separate section:

- The I/O task yields file `io.asm` as listed below:

Listing 4-6. io.asm Source File

```
          section i_o
          secflags alloc,write,noexecinstr

          global I_PORT

I_PORT

         .

         .

           source statements

         .

         .

          endsec
```

- The filter task yields file `filter.asm` as listed below:

Listing 4-7. filter.asm Source File

```
          section filter
          secflags alloc,write,noexecinstr

         .

         .

           source statements

         .
```

```
          .

          endsec
```

- The main task yields file `main.asm` as listed below:

Listing 4-8. main.asm Source File

```
          section main
          secflags alloc,write,noexecinstr

        global ENTRY

ENTRY

          .

          .

           source statements

          .

          .

          endsec
```

The next two subsections explain the methods for combining these three source files:

- Invoking a final assembly step to assign absolute addresses
- Assembling the modules separately, then linking.

4.5.1 Method 1: Absolute Mode

To assemble all project source code and assign absolute addresses, create a project file, `project.asm`, that combines the three assembly source files. The listing below shows this file.

Listing 4-9. project.asm File

```
org  PL:$1000             ; initialize entry point for program
counter
include 'main.asm'

include 'io.asm'

include 'filter.asm'

end  entry
```

To assemble file `project.asm` in absolute mode, and generate executable object file `project.eld`, use this command:

```
  scasm -a -bproject.eld project.asm
```

Upon reaching the end of file `main.asm`, the assembler returns to the next sequential statement in file `project.asm`. This statement directs the assembler to start taking input from file `io.asm`. The ORG statement in the `project.asm` file tells the assembler to set current memory space to P (program), and to initialize the L (low) location counter to $1000. This specifies assembly of the `io.asm` statements at the next available Low Program memory space.

Assembly of file `filter.asm` happens in a similar manner. The last line of file project.asm tells the assembler that it is the last logical source statement. The last line also tells the assembler that the starting address for the object module is label ENTRY. (In actual code, the ENTRY label must be declared global in section `main`.)

4.5.2 Method 2: Relative Mode

Use the assembler default mode to assemble each source file separately. Use these commands:

```
scasm -bmain.eln main.asm
```

```
scasm -bio.eln io.asm
```

```
scasm -bfilter.eln filter.asm
```

In response, the assembler generates relocatable object files `main.eln`, `io.eln`, and `filter.eln`. The assembler establishes a separate set of location counters for each input-file section. This means that all memory spaces for each section begin at relative address zero.

Use this command to invoke the linker:

```
sc3000-ld -M -c link.cmd -o project.eld main.eln io.eln filter.eln
```

In response, the linker combines the relocatable object files, reading each section's address and the program entry point from linker command file `link.cmd`. The linker processes the three `.eln` files in their command-line order, outputting executable file `project.eld`.

Chapter 5
Assembler Directives

This chapter explains the special characters significant to the assembler, as well as the directives that control assembler behavior.

In response to these directives, the assembler carries out certain actions during assembly. But it is not appropriate for the assembler to translate every directive into machine language.

NOTE

In addition to assembler directives, there are two assembler pragmas. Text at the very end of this chapter explains these directives.

In this chapter:

- Significant Characters
- Directive List
- Descriptions
- Pragmas

5.1 Significant Characters

Several one- and two-character sequences have significance for the assembler, their meanings sometimes depending on context.

The below table identifies these characters briefly; full explanations appear at the start of this chapter's reference items.

(Chapter 3 explains special characters for expression evaluation.)

Table 5-1. Assembler Significant Characters

Character	Description
;	Comment delimiter
;;	Unreported comment delimiter
\	Line continuation character; macro dummy argument concatenation operator
?	Macro value substitution operator
%	Macro hexadecimal value substitution operator
^	Macro local label override operator
"	Macro string delimiter; quoted string DEFINE expansion character
@	Function delimiter
*	Location counter substitution
++	String concatenation operator
[ ]	Substring delimiters; instruction grouping operators
<	Short addressing mode force operator
>	Long addressing mode force operator
#	Immediate addressing mode operator
#<	Immediate short addressing mode force operator
#>	Immediate long addressing mode force operator
$	Hexadecimal constants indicator
`	String constants delimiter

5.2 Directive List

This topic lists and describes the assembler directives.

The table below briefly identifies the assembler directives of each type:

- Assembly control
- Symbol definition
- Data definition/storage allocation

- Listing control and options
- Macros and conditional assembly

Table 5-2. Assembler Directive Summary

Type	Directive	Description
Assembly Control	COMMENT	Start comment lines
	DEFINE	Define substitution string
	END	End of source program
	FAIL	Programmer-generated error message
	HIMEM	Set high memory bounds
	INCLUDE	Include secondary file
	LOMEM	Set low memory bounds
	MSG	Programmer-generated message
	ORG	Initialize memory space and location counters
	SUPPRESS_ERRATUM	Suppresses errata information
	UNDEF	Undefine the DEFINE symbol
	WARN	Programmer-generated warning
Symbol Definition	ENDSEC	End section
	EQU	Equate symbol to a value
	GLOBAL	Global section symbol declaration
	GSET	Set global symbol to a value
	MULTIDEF	Allow multiple definitions
	SECFLAGS	Set ELF section flags
	SECTION	Start section
	SECTYPE	Set ELF section type
	SET	Set symbol to a value
	SIZE	Set size of symbol in the ELF symbol table
	TYPE	Set symbol type in the ELF symbol table
Data Definition and Storage Allocation	ALIGN	Align location counter
	BADDR	Set buffer address
	BSB	Block storage bit-reverse
	BSC	Block storage of constant
	BUFFER	Start buffer
	DC	Define constant
	DCB	Define constant byte
	DCL	Define constant long
	DCW	Define constant word

Table continues on the next page...

Table 5-2. Assembler Directive Summary (continued)

Type	Directive	Description
	DS	Define storage
	DSR	Define reverse carry storage
	ENDBUF	End buffer
	FALIGN	Fetch-set alignment
Listing Control and Options	LIST	List the assembly
	NOLIST	Stop assembly listing
	NOTE	Include note
	OPT	Set assembler options
	PAGE	Top of page/size page
	PRCTL	Send control string to printer
	STITLE	Initialize program subtitle
	TITLE	Initialize program title
Macros and Conditional Assembly	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
	PMARCRO	Purge macro definition

5.3 Descriptions

This subsection consists of special-character and directive descriptions, including usage guidelines and examples.

- Descriptions of special characters follow the order specified in Table 5-1 table. (The $ and `, hexadecimal and string delimiter characters, however, are so simple that they do not require such descriptions.
- Descriptions of directives are in alphabetic order, without regard the type categories of Table 5-2.

NOTE

This chapter shows directives in upper-case letters, but the assembler recognizes either case for directives.

NOTE

You may not use a label on the same line as a directive, unless the prototype includes a label parameter.

5.3.1 ; Start Comment

Starts a comment: any number of characters not part of a literal string.

Remarks

For a comment in a source-statement line, shift the ; character right, so that the comment lines up with comments of other lines. For a comment that takes up an entire line, put the ; character at the first space of the line.

Use comments to document your source program: although the assembler reproduces comments in the source listing, comments are not significant to the assembler. Macro definitions preserve comments, but you can use the NOCM option to turn off this arrangement.

Example

```
; This comment begins in column 1 of the source file

LOOP    JSR COMPUTE       ; This is a trailing comment
                          ; a source-file tab precedes
                             ; these two comments
```

5.3.2 ;; Start Unreported Comment

Starts an unreported comment: any comment that the assembler does not reproduce in the source listing, nor save in macro definitions.

Remarks

Unreported comments follow the same position rules as normal comments; you can use them to document your source program. However, such comments never appear in the assembler source listing, nor does the assembler save them in macro definitions.

Example

```
;; These lines will not be reproduced
;; in the source listing
```

5.3.3 \ Continue Line

Continues a source statement to the next line.

Remarks

If the assembler encounters a backslash (\) as the last character of a source line, it concatenates the line to the next line, processing the result as if it were a statement on one line.

(Alternate role: In a macro definition, this character concatenates a dummy argument with adjacent characters.)

Example

The \ character makes one comment span three lines:

```
; This comment \
extends over \
three lines.
```

5.3.4 \ Concatenate Macro Argument

In a macro definition, concatenates a dummy argument with adjacent characters.

Remarks

There must not be any spaces with the \ character, which can precede or follow the adjoining characters. To position an argument between two characters, use the \ character before and after the argument name.

(Alternate role: If the last character of a source line, continues the statement to the next line.)

Example

The backslash (\) in the macro definition tells the macro processor to concatenate the dummy-argument substitution characters with the character R:

```
SWAP_REG  MACRO REG1,REG2    ; Swap REG1,REG2 (D4 is temp)
          MOVE.L R\REG1,D4
          MOVE.L R\REG2,D\REG1
          MOVE.L D4,D\REG2
          ENDM
```

The macro call `SWAP_REG 0,1` results in this expansion:

```
MOVE.L R0,D4
MOVE.L R1,D0
MOVE.L D4,D1
```

5.3.5 ? Substitute Macro Value

In macro definitions, converts a `symbol` to the ASCII string that represents the symbol's decimal value. There must not be any spaces between the ? character and the `symbol`; the value of symbol must be an integer. You may use the ? character with the backslash (\) concatenation operator.

Example

Consider this macro definition:

```
SWAP_SYM  MACRO REG1,REG2      ; Swap REG1,REG2 (D4 is temp)
          MOVE.L R\?REG1,D4
          MOVE.L R\?REG2,D\?REG1
          MOVE.L D4,R\?REG2
          ENDM
```

Then suppose these SET statements and this macro call:

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

The macro processor would:

- Substitute the characters `AREG` for each occurrence of `REG1`, and `BREG` for each occurrence of `REG2`, as if producing this intermediate macro expansion:

```
MOVE.L R\?REG1,D4
MOVE.L R\?REG2,D\?REG1
MOVE.L D4,R\?REG2
```

- Replace `?AREG` with the character `0`, and `?BREG` with the character `1`, as if producing this second intermediate expansion:

```
MOVE.L R\0,D4
MOVE.L R\1,D\0
MOVE.L D4,R\1
```

- Apply the concatenation operator (\), producing the expansion that appears in the source listing:

```
MOVE.L R0,D4
MOVE.L R1,D0
MOVE.L D4,R1
```

5.3.6 % Substitute Macro Hex Value

In macro definitions, converts a `symbol` to the ASCII string that represents the symbol's hexadecimal value.

Remarks

There must not be any spaces between the % character and the `symbol`; the value of `symbol` must be an integer. You may use the % character with the backslash (\) concatenation operator.

The % character also can indicate a binary constant. If you need a binary constant inside a macro, enclose the constant in parentheses. Alternatively, follow the percent sign with a backslash (\) to escape the constant.

Example

This macro definition generates a label - the label prefix argument concatenated to a hexadecimal argument:

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

Suppose this SET statement and this macro call:

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

The macro processor would:

- Substitute the characters HEX for LAB.
- Replace %VAL with the character A (the hexadecimal equivalent of decimal 10).

- Apply the concatenation operator (\).
- Substitute the string 'NOP' for the STMT argument, producing this expansion:

```
HEXA      NOP
```

5.3.7 ^ Override Macro Local Label

If a unary expression operator in a macro expansion, specifies normal-scope (not macro-scope) evaluation for local labels in its associated term.

Remarks

If the circumflex (^) character precedes an expression term, the assembler does not search the macro local label list for any %labels in the expression term. This operator has no effect on normal labels; it has no effect at all outside a macro expansion.

The circumflex operator lets you pass local labels as macro arguments, for use as referents in the macro. Note that the circumflex also is the binary exclusive OR operator.

Example

Consider this macro definition:

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

And this macro call:

```
%LOCAL
         LOAD %LOCAL
```

The override operator tells the assembler to recognize the %LOCAL symbol outside the macro expansion, and to use that value in the MOVE instruction. If the override operator were not present, the assembler would issue an error message that %LOCAL was not defined in the macro.

5.3.8 " Delimit Macro String

In a macro definition, tells the macro processor to use a single quote ('). This transforms any enclosed dummy arguments into literal strings.

(Alternate role: In a DEFINE-directive character sequence, specifies expansion within the string.)

Example

For this macro definition:

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

And this macro call:

```
           CSTR ABCD
```

The macro expansion would be:

```
           DC 'ABCD'
```

5.3.9 " Expand DEFINE Quoted String

In a DEFINE-directive character sequence, specifies expansion within the string. (Otherwise, delimits a string, just as single quotes.)

(Alternate role: In a macro definition, tells the macro processor to use a single quote.)

Example

For this macro definition:

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

and this macro call:

```
           STR_MAC sentence
```

the macro expansion would be:

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

5.3.10 @ Start Function

Mandatory start symbol for all assembler built-in functions.

Example

```
SVAL     EQU @SQT(FVAL)        ; Obtain square root
```

5.3.11 * Substitute Location Counter

If an operand in an expression, represents the current integer value of the location counter.

Example

```
         ORG P:$100
PBASE     EQU *+$20        ; PBASE = $120
```

5.3.12 ++ Concatenate Strings

Concatenates any two strings. Single or double quotes must enclose the strings; there must not be intervening spaces.

Example

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

5.3.13 [ ] Delimit Substring

Delimit a substring operation. (Alternate role: Group instructions.)

```
[string,offset,length]
```

Parameters

`string`

Source string: any valid string combination, including another substring.

`offset`

Substring starting position within the source string, beginning at 0; may not exceed the length of the source string.

`length`

Length of the substring; may not exceed the length of the source string.

Example

```
DEFINE ID ['abcdefg',1,3]        ;ID = 'bcd'
```

5.3.14 [ ] Group Instructions

Group instructions. The opening bracket cannot appear in the label field. If the first instruction is on the same line, a space must separate the bracket and the instruction. (Alternate role: Delimit substrings.)

```
[
instruction ...
]
```

Example

```
[
 Mac.X D0.H,D1.H,D2 ; Remark1
 MAC.x D3.H,D4.H,D5
 ADD.x D0,D1,D3  ;Remark2

; remrk cont
  ld.w (R1)+,D1
]
```

5.3.15 < Force Short Addressing

Forces the assembler to use short absolute addressing, overriding default long addressing.

Remarks

Many DSP instructions permit a short form of addressing. If the assembler knows the absolute address during pass one, the assembler uses the shortest addressing mode consistent with the instruction format.

But if an address is a forward or external reference, the assembler cannot know the absolute address during pass one. Accordingly, the assembler uses the long form of addressing; this makes the instruction two words. To override this default arrangement, start the absolute address with the < character.

Example

In this sample code:

```
DATAST   EQU   $23
        tfra.l #DATAST,r28
```

The DATAST symbol is a forward reference; the assembler uses long absolute addressing: two words. To force short absolute addressing, insert the < character:

```
DATAST   EQU   $23
        tfra.l #DATAST,r28
```

5.3.16 > Force Long Addressing

Forces the assembler to use long absolute addressing.

Remarks

Many DSP instructions permit a long form of addressing. But if the assembler knows the absolute address during pass one, the assembler uses the shortest addressing mode consistent with the instruction format. To override this behavior, forcing long absolute addressing, start the absolute address with the > character.

Example

In this sample code:

```
DATAST   EQU    P:$23
          MOVE.B D0,P:DATAST
```

The DATAST symbol is *not* a forward reference; the assembler uses short absolute addressing. To force long absolute addressing, insert the > character:

```
DATAST   EQU    P:$23
          MOVE.B D0,P:>DATAST
```

5.3.17 # Use Immediate Addressing

Tells the assembler to use immediate addressing mode.

Example

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

5.3.18 #< Force Immediate Short Addressing

Forces the assembler to use immediate short addressing, overriding default immediate long addressing.

Remarks

Many DSP instructions permit a short immediate form of addressing. If the assembler knows the immediate data during pass one, the assembler uses the shortest addressing mode consistent with the instruction format.

But if the immediate data is a forward or external reference, the assembler cannot know the immediate data during pass one. Accordingly, the assembler uses the long form of immediate addressing; this makes the instruction two words. To override this default behavior, start the immediate data symbol with the #< characters.

Example

For this sample code:

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

The assembler does not know the `CNST` symbol during pass one; the assembler uses immediate long addressing: two words. To force immediate short addressing, insert the #< characters:

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

5.3.19 #> Force Immediate Long Addressing

Forces the assembler to use immediate long addressing.

Remarks

Many DSP instructions permit a long immediate form of addressing. But if the assembler knows the immediate data during pass one, the assembler uses the shortest addressing mode consistent with the instruction format. To override this behavior, forcing immediate long addressing, start the immediate data symbol with the #> characters.

Example

For this code:

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

The DATAST symbol is *not* a forward reference; the assembler uses short absolute addressing. To force long absolute addressing, insert the > character:

The assembler knows the CNST symbol during pass one; the assembler uses immediate short addressing. To force immediate long addressing, insert the #> characters:

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

5.3.20 ALIGN Align Location Counter

Advances the location counter, aligning it on the specified address boundary. If the location counter already is aligned on this boundary, this directive has no effect.

```
ALIGN boundary
```

Parameter

`boundary`

Address boundary specifier; must be a power of two.

Example

```
ALIGN 4         ; Align location counter to
                 ; next long word boundary
```

5.3.21 BADDR Set Buffer Address

Sets the location counter to the address of a reverse-carry buffer. Does not initialize the block of memory intended for the buffer.

```
BADDR R,length
```

Parameters

`R`

Specifier for the reverse carry buffer type.

`length`

Buffer length, in bytes: an expression that evaluates to an absolute integer greater than zero. Must not contain any forward references; should be a power of two.

Remarks

If the location counter value is not zero, this directive advances the location counter to a base address that is a multiple of 2k, where:

```
2k Š length
```

There must be sufficient remaining memory to establish a valid base address, or the assembler issues an error message. Unlike other buffer allocation directives, the BADDR directive does *not* advance the location counter; the location counter continues to point to the buffer base address.

You may not use a label with this directive. The assembler issues a warning if the `length` value is not a power of two.

Related Directives

BSB, BUFFER, DSR

Example

```
ORG P:$100
BADDR R,24        ; Reverse buffer 24
```

5.3.22 BSB Allocate Bit-Reverse Buffer

Allocates and initializes a block of bytes for a reverse-carry buffer.

```
[label] BSB length[,value]
```

Parameters

`label`

Optional label that receives the value of the location counter, once the assembler establishes a valid base address.

`R`

Specifier for the reverse carry buffer type.

`length`

Block length, in bytes: an expression that evaluates to an absolute integer greater than zero. Must not contain any forward references; should be a power of two. Can have any memory space attribute.

`value`

Optional value expression for the initial value of each array byte. Can have any memory space attribute. Omitting this value tells the assembler to use the value zero.

Remarks

If the location counter is not zero, this directive advances the location counter to a base address that is a multiple of 2k, where:

```
2k Š length
```

The listing shows only one byte of object code, regardless of how large the `length` expression is. However, the location counter advances by the number of bytes generated.

The assembler issues a warning if the `length` expression is not a power of two.

Related Directives

BADDR, BSC, DC

Example

```
BUFFER BSB BUFSIZ        ; Initialize buffer to zeros
```

5.3.23 BSC Allocate Constant Storage Block

Allocates and initializes a block of bytes.

```
[label] BSC length[,value]
```

Parameters

`label`

Optional label that receives the value of the location counter at the start of directive processing.

`length`

Block length, in bytes: an expression that evaluates to an absolute integer greater than zero. Must not contain any forward references. Can have any memory space attribute.

`value`

Optional value expression for the initial value of each block byte, in the range -128..+255. Can have any memory space attribute. Omitting this value tells the assembler to use the value zero.

Remarks

The listing shows only one byte of object code, regardless of how large the `length` expression is. However, the location counter advances by the number of bytes generated.

Related Directives

BADDR, BSB, DC

Example

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

5.3.24 BUFFER Start Buffer

Indicates the beginning address of a reverse-carry buffer; does not initialize .

```
BUFFER R,length
```

Parameters

`R`

Specifier for the reverse carry buffer type.

`length`

Buffer length, in bytes: an expression that evaluates to an absolute integer greater than zero. Must not contain any forward references; should be a power of two.

Remarks

In response to this directive, the assembler allocates data for the buffer until it encounters an ENDBUF directive. If allocated data does not fill the buffer, unfilled locations remain uninitialized; if allocated data exceeds the specified buffer size, the assembler issues an error message.

Instructions and most data definition directives may appear between the BUFFER and ENDBUF directive pair. But you must not nest BUFFER directives. The directives that may not appear between BUFFER and ENDBUF are:

- MODE
- ORG
- SECTION
- Other buffer allocation directives

The BUFFER directive sets the location counter to the address of a buffer of the given type. If the location counter is not zero, this directive advances the location counter to a base address that is a multiple of 2k, where:

```
2k Š length
```

There must be sufficient remaining memory to establish a valid base address, or the assembler issues an error message. Unlike other buffer allocation directives, the BUFFER directive does *not* advance the location counter; the location counter continues to point to the buffer base address.

You may not use a label with this directive. The assembler issues a warning if the `length` value is not a power of two.

Related Directives

BADDR, BSB, DSR, ENDBUF

Example

```
         ORG P:$100
         BUFFER R,28             ; Reverse buffer 28
R_BUF    DC 0.5,0.5,0.5,0.5
         DS 20                   ; Remainder uninitialized
          ENDBUF
```

5.3.25 COMMENT Start Comment Lines

Defines one or more lines as comments.

```
COMMENT delimiter
.
.
delimiter
```

Parameter

`delimiter`

Any non-space character.

Remarks

The two delimiter characters define comment text. The line that contains the second delimiter character is the last line of the comment. Comment text can include any printable characters; the assembler does reproduce this text in the source listing.

You may not use a label with this directive.

Examples

```
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.
         *
```

5.3.26 DC Define Constant

Allocates and initializes two bytes of memory for each argument.

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

Parameters

`label`

Optional label that receives the value of the location counter at the start of directive processing.

`arg`

An integer constant, fractional constant, symbol, or expression. Commas without spaces must separate multiple `arg` values.

Remarks

The assembler stores multiple arguments in successive address locations. If this directive has multiple arguments, one or more can be null (two adjacent commas): this fills the corresponding address location with zeros.

If you use the DC directive in L memory, the assembler evaluates and stores arguments as long word quantities. Otherwise, the assembler issues an error message if the evaluated argument value is too large for a single DSP word.

The assembler stores integer arguments as integers; it converts floating-point arguments to binary values. String storage is:

- Single-character strings: a word whose lower seven bits represent the ASCII character value. For example, the assembler stores the string `'R' as $000052.`
- Multiple-character strings: words that are concatenated sequences of ASCII values. If the number of string characters is not an even multiple of the number of bytes per DSP word, the last word's remaining characters are left aligned, and zeros fill the rest of the word. For example, the assembler stores the string `ABCD' as two words, $414243 and $410000.
- Exception: The NOPS option tells the assembler to store each character of a string as it would store a single-character string. It would store the string `ABCD' as $000041, $000042, $000043, and $000044.

Related Directives

DCB, DCL

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

5.3.27 DCB Define Constant Byte

Allocates and initializes a byte of memory for each argument.

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

Parameters

`label`

Optional label that receives the value of the location counter at the start of directive processing.

`arg`

A byte integer constant, string constant, symbol, or byte expression. Integer constants must be byte values, in the range 0 - 255. Commas without spaces must separate multiple `arg` values.

Remarks

The assembler stores multiple arguments in successive byte locations. If this directive has multiple arguments, one or more can be null (two adjacent commas): this fills the corresponding byte location with zeros.

The assembler stores integer arguments as integers; you may not use floating-point arguments. String storage is:

- Single-character strings: a byte whose lower seven bits represent the ASCII character value. For example, the assembler stores the string `'R' as $52.`
- Multiple-character strings: consecutive bytes, each of which contains an ASCII value. For example, the assembler stores the arguments `AB',,'CD' as $41, $42, $00, $43, and $44.

Related Directives

DC, DCL, DCLL

Example

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

5.3.28 DCL Define Constant Long

Allocates and initializes four bytes of memory for each argument.

```
[label]DCL arg[,arg,...]
```

Parameters

`label`

Optional label that receives the value of the runtime location counter at the start of directive processing.

`arg`

An integer constant, fractional constant, symbol, or expression. Commas without spaces must separate multiple `arg` values.

Related Directives

DC, DCB

Example

```
DCL $12345678      ; only big endian mode in sc3900fp,
                   ; $12345678 = $12
                   ;             $34
                   ;             $56
                   ;             $78
```

5.3.29 DCLL Define Constant Long Long

Allocates and initializes the eight bytes of memory for each argument.

```
[label]DCL arg[,arg,...]
```

Parameters

`label`

Optional label that receives the value of the runtime location counter at the start of directive processing.

`arg`

An integer constant, fractional constant, symbol, or expression. Commas without spaces must separate multiple arg values.

Related Directives

DC, DCB, DCL

Example

```
DCLL $0123456789ABCDEF
```

5.3.30 DEFINE Define Substitution String

Defines substitution strings that the assembler uses in all following source lines.

```
DEFINE symbol string
```

Parameters

`symbol`

Valid global or local symbol that appears in source lines.

`string`

Replacement string for the symbol.

Remarks

Upon encountering a macro definition, the assembler applies DEFINE-directive translations. Later, when the assembler expands the macro, it applies the appropriate DEFINE-directive translations to the expansion.

You may not use a label with this directive.

Related Directives

GSET, SET, UNDEF

Example

```
          DEFINE ARRAYSIZ '16*SAMPLSIZ'
SAMPLSIZ  EQU    16
          DS     ARRAYSIZ    ; This line transformed to
          .                  ; DS 16*SAMPLSIZ
          .
```

5.3.31 DS Define Storage

Reserves a block of bytes in memory, but does not initialize the block.

```
[label]  DS numbytes
```

Parameters

`label`

Optional label that receives the value of the location counter at the start of directive processing.

`numbytes`

Number of bytes: an expression that evaluates to an integer greater than zero. This expression must not contain any forward references. The location counter advances by this number of bytes.

Related Directive

DSR

Example

```
; If the current loader address is $9e


      align16; Align on next 16-byte boundary
R_BUF DS    8    ; Reserve 8 bytes for R_BUF
S_BUF DS     12    ; Reserve 12 bytes for S_BUF
```

5.3.32 DSR Define Reverse-Carry Storage

Reserves a block of bytes in memory for a reverse-carry buffer, but does not initialize the block.

```
[label] DSR numbytes
```

Parameters

`label`

Optional label that receives the value of the location counter, once the assembler establishes a valid base address.

`numbytes`

Number of bytes: an expression that evaluates to an absolute integer greater than zero. This expression should be a power of two, and must not contain any forward references.

Remarks

This directive advances the location counter from a valid base address:

- If the location-counter value is not zero, the assembler advances the location counter to a base address that is a multiple of 2,k where 2k Š `numbytes`. The assembler issues an error message if there is insufficient memory.
- Then the assembler advances the location counter by the value of `numbytes`.

The assembler generates a warning if `numbytes` is not a power of two.

Related Directive

DS

Example

```
      ORG P:$100 ; Set address to P:$100
R_BUF DSR 8        ; Reserve 8 bytes for R_BUF
```

5.3.33 DUP Duplicate Source Lines

Duplicates the following source lines the specified number of times; the ENDM directive marks the last line to be duplicated.

```
[label] DUP times
         .
         .
          ENDM
```

Parameters

`label`

Optional label that receives the value of the runtime location counter at the start of directive processing.

`times`

Number of times to duplicate source lines: an expression that evaluates to an absolute integer. This expression can have any memory-space attribute, but must not contain any forward references. If the times value is less than or equal to zero, the assembler output does not include the sequence of lines.

Remarks

You may nest the DUP directive to any level. If the `times` value is less than or equal to zero, the assembler output does not include the sequence of lines.

To immediately halt source-line duplication, for example, upon detection of an error, use the EXITM directive and conditional-assembly directives.

Related Directives

DUPA, DUPC, DUPF, EXITM, MACRO

Example

If MD and MEX options are enabled, and if input includes these lines:

```
COUNT SET   3
      DUP   COUNT    ; ASR BY COUNT
      ASH.RGT.X #1,D0,D0
      ENDM
```

The source listing includes the line `ASR D0` three times.

5.3.34 DUPA Duplicate Sequence with Arguments

Duplicates the following source statements for each argument, substituting successive `arg` values for the `dummy` argument. The ENDM directive marks the last statement to be duplicated. To immediately halt source-line duplication, for example, upon detection of an error, use the EXITM directive and conditional-assembly directives.

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

Parameters

`label`

Optional label that receives the value of the runtime location counter at the start of directive processing.

`dummy`

Valid expression that appears in the source lines.

`arg`

Argument string. Single quotes must enclose an embedded space or other character significant to the assembler. If a null string, the assembler removes `dummy` values as it repeats the statements. Commas without spaces must separate multiple `arg` values.

Related Directives

DUP, DUPC, DUPF, ENDM, EXITM, MACRO

Example

If MD and MEX options are enabled, and if input includes these lines:

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

The source listing shows successive lines:

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

5.3.35 DUPC Duplicate Sequence with Characters

Duplicates the following source statements for each character of the string argument, substituting successive `string` characters for the `dummy` argument. The ENDM directive marks the last statement to be duplicated. To immediately halt source-line duplication, for example, upon detection of an error, use the EXITM directive and conditional-assembly directives.

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

Parameters

`label`

Optional label that receives the value of the runtime location counter at the start of directive processing.

`dummy`

Valid expression that appears in the source lines.

`String`

Valid string expression.Argument string. Single quotes must enclose an embedded space or other character significant to the assembler. If a null string, the assembler skips the block of statements.

Related Directives

DUP, DUPA, DUPF, ENDM, EXITM, MACRO

Example

If MD and MEX options are enabled, and if input includes these lines:

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

The source listing shows successive lines:

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

5.3.36 DUPF Duplicate Sequence in Loop

Duplicates the following source statements; the values of `start`, `end`, and `increment` arguments determine the number of duplications. The ENDM directive marks the last statement to be duplicated. To immediately halt source-line duplication, for example, upon detection of an error, use the EXITM directive and conditional-assembly directives.

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

Parameters

`label`

Optional label that receives the value of the runtime location counter at the start of directive processing.

`dummy`

A parameter that holds the loop index value; may appear in instruction statements.

`start`

Optional starting value for the loop index; defaults to 1.

`end`

Ending value for the loop index.

`increment`

Optional increment value for the loop index; defaults to 1.

Related Directives

DUP, DUPA, DUPC, ENDM, EXITM, MACRO

Example

If MD and MEX options are enabled, and if input includes these lines:

```
DUPF   NUM,0,7
ST.L D0,(R\NUM)
ENDM
```

The source listing includes eight copies of the line `MOVE.B #0,R\NUM`.

5.3.37 ELSE Start Alternative Conditional Assembly

Delimits alternative conditional assembly: ends or begins source lines to be assembled, according to the value of the assembly condition.

```
ELSE
```

Remarks

An optional directive valid only with a pair of IF and ENDIF directives:

- If the IF directive's `condition` value is TRUE (non-zero), the assembler assembles source lines between the IF and ELSE directives, ignores lines between the ELSE and ENDIF directives.
- If the IF directive's `condition` value is FALSE (zero), the assembler ignores source lines between the IF and ELSE directives, assembles lines between the ELSE and ENDIF directives.

You can nest conditional directives to any level. The ELSE directive, like the ENDIF directive, always pairs with the closest previous IF directive. You may not use a label with this directive.

Related Directives

ENDIF, IF

Example

If `FLOW_1` was defined and the assembler encounters this code:

```
IF @DEF('FLOW_1')

LD.W (r0)-,d2         ; Start traceback from
ELSE                       ; state zero.
  LD.W (r0),d1
ENDIF
```

The assembler ignores the line `LD.W (r0),d1` and assembles the line `LD.W (r0)-,d2`.

5.3.38 END End of Source Program

Marks that the logical end of the source program; the assembler ignores any statements following this directive.

```
END [startaddr]
```

Parameter

startaddr

Optional starting execution address of the program. Only valid for absolute mode; must have memory-space attribute P (program) or N (none).

Remarks

You cannot use this directive in a macro expansion. You may not use a label with this directive.

Example

```
END BEGIN          ; BEGIN is the starting execution

address
```

5.3.39 ENDBUF End Buffer

Marks the end of a buffer block.

```
ENDBUF
```

Remarks

When the assembler encounters this directive, the location counter points just beyond the end of the buffer. You may not use a label with this directive.

Related Directive

BUFFER

Example

```
ORG P:$100
BUFFER R,64        ; Uninitialized reverse-carry buffer
ENDBUF
```

5.3.40 ENDIF End Conditional Assembly

Ends conditional assembly that the preceding IF directive began.

```
ENDIF
```

Remarks

The directive pair IF and ENDIF delimit source lines for conditional assembly. You can nest conditional assembly directives to any level. The ENDIF directive, like the optional ELSE directive, always pairs with the closest previous IF directive.

You may not use a label with this directive.

Related Directives

ELSE, IF

Example

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

5.3.41 ENDM End Macro Definition

Ends a macro definition or marks the end of duplicated lines: terminates assembly actions of the MACRO, DUP, DUPA, DUPC, or DUPF directives. You may not use a label with this directive.

```
ENDM
```

Related Directives

DUP, DUPA, DUPC, DUPF, MACRO

Example

```
SWAP_SYM   MACRO REG1,REG2     ; Swap REG1,REG2 (D4 is temp)
           MOVE.L R\?REG1,D4
           MOVE.L R\?REG2,D\?REG1
           MOVE.L D4,R\?REG2
           ENDM
```

5.3.42 ENDSEC End Section

Marks the end of a section; the previous SECTION directive began the section. You may not use a label with this directive.

```
ENDSEC
```

Related Directive

SECTION

Example

```
         SECTION .data
VALUES   BSC $100        ; Initialize to zero
          ENDSEC
```

5.3.43 EQU Equate Symbol to Value

Assigns an expression value to a label symbol.

```
label   EQU [L:|N:|P:]expression
```

Parameters

`label`

Label that receives the `expression` value.

`L:|N:|P:`

Optional specifier that forces the memory-space attribute; valid only if the expression memory-space attribute is N (none).

`expression`

Any absolute or relative expression; must not include any forward references.

Remarks

Many directives assign the program-counter value to a label; this directive gives an expression value to the specified label. You cannot redefine this label anywhere in the program or section.

The optional forcing memory space parameter lets you assign an attribute to a constants-only expression that refers to a fixed address in a memory space. However, if the `expression` attribute is L or P and you specify an attribute that does not match, the assembler issues an error message.

Examples

This first example assigns the value $4000, and memory-space attribute P, to the symbol `A_D_PORT`:

```
A_D_PORT  EQU P:$4000
```

This second example gives symbol `COMPUTE` the value and memory-space attribute of the expression `@LCV(L)`:

```
COMPUTE   EQU @LCV(L)
```

5.3.44 EXITM Exit Macro

Immediately terminates a macro expansion or a sequence of duplicated lines.

```
EXITM
```

Remarks

Use this directive with conditional-assembly directives to terminate macro expansion (or duplicated lines) upon detection of an error condition. You may not use a label with this directive.

Related Directives

DUP, DUPA, DUPC, DUPF, MACRO

Example

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

5.3.45 FAIL Issue Programmer Error Message

Issues the specified error messages and increments the total error count.

```
FAIL {str | exp}[,{str | exp},...]
```

Parameters

`str`

Any valid string appropriate as part of an error message. Commas without spaces must separate multiple `str` values.

`exp`

Any expression appropriate as part of an error message. Commas without spaces must separate multiple `exp` values.

Remarks

Use this directive with conditional assembly directives for exceptional condition checking. The assembly proceeds normally after the assembler displays the error message.

You may not use a label with this directive.

Related Directives

MSG, WARN

Example

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

5.3.46 FALIGN Align with Fetch-Set

Aligns the address of a hardware loop's first instruction, or the address of a jump-instruction target, with the fetch set. The fetch set boundary is 32 bytes for the SC3900FP core.

```
FALIGN
```

Remarks

You may apply the FALIGN directive on a per-loop basis. The assembler performs alignment if the size of the execution set overlaps the fetch set boundary. If the execution set starts at a nonaligned address, but fits into the current fetch set, the assembler does not perform alignment.

The assembler implements alignment by padding:

- If you optimize the function for speed (OPT_SPEED - the default), the assembler inserts the appropriate number of NOPs inside packets, before the loop that contains the FALIGN directive.
- If you optimize the function for size (OPT_SIZE), the assembler inserts the appropriate number of NOPs as a standalone packet.

Any time the assembler inserts NOPS, it issues a remark. (The `-ofa` and `-onofa` command-line options enable/disable display of these remarks.)

For the SC3900FP core, the FALIGN directive forces its entire section to a 32-byte alignment. The system preserves these alignments at link time, even if the section starting location has moved.

Related Directive

OPT (LPA option)

Example

```
        ORG P:$100
        DOEN.3 #5
        NOP
        NOP
        NOP
        LOOPSTART3
         FALIGN

compute_alpha
         ...
          LOOPEND3
```

5.3.47 GLOBAL Declare Global Section Symbol

Makes specified local section symbols global. (The default arrangements for such symbols is that they are local.) You may not use a label with this directive.

```
GLOBAL symbol[,symbol,...]
```

Parameter

`symbol`

Any symbol defined within the section, that is, between the SECTION and ENDSEC directives. Commas without spaces must separate multiple `symbol` values.

Related Directive

SECTION

Example

```
SECTION IO
GLOBAL LOOPA      ; LOOPA now globally accessible
.                 ; by other files
.
.
ENDSEC
```

5.3.48 GSET Set Global Symbol to Value

Assigns the specified value to the specified global symbol.

```
label     GSET value
          GSET label value
```

Parameters

`label`

A global symbol.

`value`

An absolute expression. Must not contain any forward references.

Remarks

If you use this directive to define a label, another GSET or SET directive elsewhere in the program can redefine the label. Use this directive to reset a global SET symbol within a section, where the SET symbol otherwise would be local.

Related Directives

DEFINE, EQU, SET

Example

```
COUNT   GSET 0         ; Initialize count
```

5.3.49 HIMEM Set High Memory Bounds

Establishes an absolute high memory bound for code and data generation. This directive is valid only for absolute mode; you may not use a label with this directive.

```
HIMEM P:expr[,...]
```

Parameters

`P:`

P memory specifier.

`expr`

An absolute integer value within the processor address range.

Related Directive

LOMEM

Example

```
HIMEM P:$7FFF        ; Set P run high mem limit
 to $7fff
```

5.3.50 IF Start Conditional Assembly

Starts conditional assembly of source lines: the assembler assembles the following lines if the `condition` value is true (non-zero). The ENDIF directive indicates the last source line to be assembled conditionally.

```
IF condition
.
.
[ELSE]
.
.
ENDIF
```

Parameter

`condition`

An absolute-integer expression. A non-zero value means that the condition is true; a zero value means that the condition is false. Must not contain any forward references.

Remarks

The directive pair IF and ENDIF delimit source lines for conditional assembly. You can nest conditional assembly directives to any level. The ENDIF directive, like the optional ELSE directive, always pairs with the closest previous IF directive.

The assembler follows these rules:

- `condition` = TRUE (non-zero), no ELSE directive - assembles lines between IF and ENDIF.
- `condition` = FALSE (zero), no ELSE directive - ignores lines between IF and ENDIF.
- `condition` = TRUE (non-zero), ELSE directive present - assembles lines between IF and ELSE, ignores lines between ELSE and ENDIF.
- `condition` = FALSE (zero), ELSE directive preset - ignores lines between IF and ELSE, assembles lines between ELSE and ENDIF.

You may not use a label with this directive.

Related Directives

ELSE, ENDIF

Example

If `FLOW_1` was defined and the assembler encounters this code:

```
IF @DEF('FLOW_1')


LD.W (r0)-,d2         ; Start traceback from
ELSE                       ; state zero.
  LD.W (r0),d1
ENDIF
```

The assembler ignores the line `LD.W (r0),d1` and assembles the line `LD.W (r0)-,d2`.

5.3.51 INCLUDE Include Secondary File

Tells the assembler to read source statements from the secondary file that the `string` argument identifies.

```
INCLUDE {'file' | <file>}
```

Parameters

`'file' or <file>`

A file name compatible with the operating system; may include a pathname. The default extension is .asm.

Remarks

The assembler's search path depends on the file name syntax. For the `'file'` syntax, the assembler first searches in the current file (or in the specified directory, if the file value includes a pathname). If it does not fine the file, it next searches in all directories that the `-i` command-line option specifies.

For the `<file>` syntax, the assembler ignores the current (or specified) directory, searching only in the directories that the `-i` command-line option specifies.

You may not use a label with this directive.

Related Directive

MACLIB

Example

```
INCLUDE 'headers/io.asm'     ; Include file io.asm,
                                 ; in directory headers.

INCLUDE <data>               ; Include file data.asm,
                               ; but do not search in the
                                 ; current directory
```

5.3.52 LIST List Assembly

Prints the source listing: all lines that follow the LIST directive.

```
LIST
```

Remarks

The printed source listing begins with the line after the LIST directive.

A special list counter, initialized to 1, affects list printing: as long as the counter value remains above zero, the assembler prints the source listing in response to any LIST directive. Each LIST directive increments the list counter, but each NOLIST directive decrements the counter. If the counter value drops to zero or below, the assembler does not print a listing in response to the LIST directive.

If the command line includes the `-oIL` option, the assembler ignores the LIST directive, regardless of the counter value.

You may not use a label with this directive.

Related Directives

NOLIST, OPT

Example

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

5.3.53 LOMEM Set Low Memory Bounds

Establishes an absolute low memory bound for code and data generation. This directive is valid only for absolute mode; you may not use a label with this directive.

```
LOMEM P:expr[,...]
```

Parameters

`P:`

P memory specifier.

`expr`

An absolute integer value within the processor address range.

Related Directive

HIMEM

Example

```
HIMEM P:$100        ; Set P run low mem limit to $100
```

5.3.54 MACLIB Specify Macro Library

Specifies a directory that contains macro definitions.

```
MACLIB pathname
```

Parameter

`pathname`

Pathname of a macro-definition directory.

Remarks

Each macro definition must be in a separate file; the file must have the same name as the macro, with extension `.asm`. So file `blockmv.asm` would contain the definition of macro `blockmv`.

Upon encountering a directive that is not in the directive or mnemonic tables, the assembler searches the directory that `pathname` specifies. If it finds a matching file, the assembler saves the current source line, then opens the file for input as an INCLUDE file. When the assembler reaches the end of the file, it restores the source line and resumes processing.

If the processed file does not include a macro definition of the unknown directive, the assembler issues an error message upon restoring and processing the source line. However, the processed file is not limited to macro definitions; it can include any valid source code statements.

If multiple MACLIB directives specify several directories, the assembler searches the directories in their order in the MACLIB directives.

You may not use a label with this directive.

Related Directive

INCLUDE

Example

```
MACLIB 'macros/mymacs/'
```

5.3.55 MACRO Define Macro

Defines a new macro.

```
label     MACRO [dumarg[,dumarg,...]]
          .
          .
          source statements
          .
          .
           ENDM
```

Parameters

`label`

Name for the new macro; should not duplicate any existing assembler directives or mnemonics.

`dumarg`

Symbolic name to be replaced by an argument value when a statement calls the macro. Must follow the rules for symbol names; may not begin with the % character. Commas without spaces must separate multiple dumarg values.

Remarks

Each macro definition consists of three parts:

- Header - the name and dummy arguments. This is the MACRO directive, with its `label` and `dumarg` values. (The assembler issues a warning if the label value duplicates an assembler directive or mnemonic.)
- Body - the sequence of standard source statements.
- Terminator - the ENDM directive.

If you nest macro definitions, the assembler does not define the nested macro until it expands the primary macro.

Related Directives

DUP, DUPA, DUPC, DUPF, ENDM

Example

```
SWAP_SYM  MACRO REG1,REG2              ; Swap REG1,REG2 (D4 is
temp)
          MOVE.L R\?REG1,D4
          MOVE.L R\?REG2,D\?REG1
          MOVE.L D4,R\?REG2
          ENDM
```

5.3.56 MSG Issue Programmer Message

Outputs a message without incrementing the error or warning counts.

```
MSG {str | exp}[,{str | exp},...]
```

Parameters

`str`

Any valid string appropriate as part of a message. Commas without spaces must separate multiple `str` values.

`exp`

Any expression appropriate as part of a message. Commas without spaces must separate multiple `exp` values.

Remarks

Use this directive with conditional assembly directives to convey information. The assembly proceeds normally after the assembler displays the message.

You may not use a label with this directive.

Related Directives

FAIL, WARN

Example

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

5.3.57 MULTIDEF Allow Multiple Definitions

Allows multiple definitions for specified symbols.

```
MULTIDEF symbol[, symbol,...]
```

Parameter

`symbol`

A symbol name. Commas without spaces must separate multiple `symbol` values.

Remarks

Binds local symbols, so that they behave like global symbols. Any time the assembler combines two such bound symbols into an .eld file, it uses the first definition.

Example

```
MULTIDEF AxisX, AxisY, AxisZ
```

5.3.58 NOLIST Stop Assembly Listing

Stops printing of the assembly listing.

```
NOLIST
```

Remarks

A special list counter, initialized to 1, affects list printing: as long as the counter value remains above zero, the assembler prints the source listing in response to any LIST directive. Each LIST directive increments the list counter, but each NOLIST directive decrements the counter. If the counter value drops to zero or below, the assembler does not print a listing in response to the LIST directive.

You may not use a label with this directive.

Related Directives

LIST, OPT

Example

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

5.3.59 NOTE Include Note

Tells the assembler to include the specified note in the .note section of the listing file.

```
NOTE"<comment>"
```

Parameter

`<comment>`

User-specified note or comment.

Example

```
NOTE"This is a note"
```

5.3.60 OPT Set Assembler Options

Specifies options that control formats, reporting, and other aspects of assembler operation.

```
OPT option[,option...]      [; comment]
```

Parameters

`option`

Any assembler control option: valid either with the OPT directive or with the -o command line option. The table given below lists these options.

`comment`

Optional comment string.

Remarks

Commas without spaces must separate multiple options. You may not use a label with this directive.

The same options can be arguments for this directive or for the `-o` command line option: `OPT MU` in a source line has the same effect as `-oMU` in the command line.

The below table lists available options, by type. The table shows the options in capital letters, but the options are not case sensitive. Many options have logical opposites that begin with NO; table explanations include these opposites.

NOTE

All options available in the below table (and of Table 2-1 table) are valid with the `-Xasm` passthrough option of the `scc` command line.

Table 5-3. OPT Options

Option	Control Action
Listing Format Control	
FC	Folds trailing comments under the source line, aligning them with the opcode field; aligns lines that start with the ; character with the label field. Opposite option: NOFC, which is the default, as well as the reset setting at the end of pass one.
FF	Uses form feeds for page ejects in the source listing.Opposite option: NOFF, which is the default, as well as the reset setting at the end of pass one.
FM	Formats messages: aligns text, breaks at word boundaries.Opposite option: NOFM, which is the default, as well as the reset setting at end of pass one.
NOFC	Does not fold trailing comments under the source line. Default setting, as well as the reset setting after pass one.Opposite option: FC.
NOFF	Uses multiple line feeds for page ejects. Default setting, as well as the reset setting at the end of pass one.Opposite option: FF.
NOFM	Does not format messages. Default setting, as well as the reset setting after pass one.Opposite option: FM.

Table continues on the next page...

Table 5-3. OPT Options (continued)

Option	Control Action
NOPP	Does not pretty print; preserves source-line formatting, but expands tabs to spaces and concatenates continuation lines.Opposite option: PP, which is the default setting, as well as the reset setting after pass one.
NORC	Does not use relative spacing. Default setting, as well as the reset setting after pass one.Opposite option: RC.
PP	Pretty prints: aligns values in consistent columns, without regard to source-file formatting. Default setting, as well as the reset setting at end of pass one.Opposite option: NOPP.
RC	Uses relative comment spacing: lets comment starting column float, according to presence/absence of other field values.Opposite option: NORC, which is the default, as well as the reset setting at end of pass one.
Output file format	
REL	Specifies a .rel relocation type.
RELA	Specifies a .rela relocation type.
ELF	Specifies an ELF binary output file.
Reporting	
CEX	Prints DC expansions.Opposite option: NOCEX, which is the default, as well as the reset setting at end of pass one.
CL	Prints conditional assembly directives. Default setting, as well as the reset setting at end of pass one.Opposite option: NOCL.
DXL	Expands DEFINE directive strings in the source listing. Default setting, as well as the reset setting at end of pass one.Opposite option: NODXL.
HDR	Generates listing header along with titles and subtitles. Default setting, as well as the reset setting at end of pass one. Opposite option: NOHDR.
IL	Inhibits (blocks) a source listing.
MC	Prints macro calls. Default setting, as well as the reset setting at end of pass one.Opposite option: NOMC option: Does not print macro calls.
MD	Prints macro definitions. Default setting, as well as the reset setting at end of pass one.Opposite option: NOMD.
MEX	Prints macro expansions.Opposite option: NOMEX, which is the default setting, as well as the reset setting at end of pass one.
MU	Includes a memory utilization report in the source listing. This option must appear before any code or data generation.
NL	Prints nesting levels in listing: conditional assembly and section nesting.Opposite option: NONL, which is the default setting, as well as the reset setting at end of pass one.
NOCEX	Does not print DC expansions. Default setting, as well as the reset setting at end of pass one.Opposite option: CEX.

Table continues on the next page...

Table 5-3. OPT Options (continued)

Option	Control Action
NOCL	Does not print conditional assembly directives.Opposite option: CL, which is the default, as well as the reset setting at end of pass one.
NODXL	Does not expand DEFINE directive strings.Opposite option: DXL, which is the default, as well as the reset setting at end of pass one.
NOHDR	Does not generate listing header; turns off titles and subtitles.Opposite option: HDR, which is the default, as well as the reset setting at end of pass one.
NOMC	Does not print macro calls.Opposite option: MC, which is the default, as well as the reset setting at end of pass one.
NOMD	Does not print macro definitions.Opposite option: MD, which is the default, as well as the reset setting at end of pass one.
NOMEX	Does not print macro expansions. Default setting, as well as the reset setting at end of pass one.Opposite option: MEX.
NONL	Does not print nesting levels. Default setting, as well as the reset setting at end of pass one.Opposite option: NL.
NOU	Does not print unassembled, conditional-assembly lines. Default setting, as well as the reset setting at end of pass one.Opposite option: U.
WEX	Counts warnings as error messages. Warnings block creation of an object file, unless `-osvo` is in the command line.
U	Prints unassembled lines of conditional-assembly code.Opposite option: NOU, which is the default, as well as the reset setting at end of pass one.
Message Control	
AE	Checks address expressions: validates arithmetic operations. Default setting, as well as the reset setting at end of pass one.Opposite option: NOAE.
FA	Enables display of FALIGN-directive remarks.Opposite option: NOFA, which is the default.
MSW	Issues a memory space warning if incompatibilities exist. Default setting, as well as the reset setting at end of pass one.Opposite option: NOMSW.
NOAE	Does not check address expressions.Opposite option: AE, which is the default, as well as the reset setting at end of pass one.
NOFA	Disables display of FALIGN-directive remarks. Default setting.Opposite option: FA.
NOMSW	Suppresses memory space incompatibility warnings.Opposite option: MSW, which is the default, as well as the reset setting at end of pass one.
NOR	Disables remarks display. (Disabling warnings display also blocks display of remarks.)Opposite option: R, which is the default.
NOSTALLS	Specifies no generation of SC3000 stall information. Default setting; not valid for other architectures.Opposite option: STALLS.

Table continues on the next page...

Table 5-3. OPT Options (continued)

Option	Control Action
NOUR	Does not warn about unresolved references.Oppsite option: UR (which is valid only in relocatable mode).
NOW	Does not print warnings.Opposite option: W, which is the default, as well as the reset setting at end of pass one.
R	Enables remarks display. Default setting.Opposite option: NOR.
STALLS	Specifies generation of SC3000 stall information in listing files; not valid for other architectures.NOTE: Stall information appears as warnings, so you must enable warnings display to see this information.Opposite option: NOSTALLS, which is the default.
UR	Warns about each unresolved reference at assembly time; valid only in relocatable mode.Opposite option: NOUR.
W	Prints all warnings. Default setting, as well as the reset setting at end of pass one.Opposite option: NOW.
Symbol	
DEX	Expands DEFINE symbols in quoted strings. (To expand individual symbols, use double-quoted strings.)
SO	Writes symbol information to the object file.
Assembler Operation	
BE	Generates output for a big-endian target. (The default setting is little-endian.)
CC	Enables cycle counts, clears the total cycle count. The output listing shows cycle counts for each instruction, as if there were a full instruction fetch pipeline and no wait states.Opposite option: NOCC, which is the default, as well as the reset setting at end of pass one.
CK	Enables checksumming for instructions and data values; clears cumulative checksum.Opposite option: NOCK, which is the default, as well as the reset setting at end of pass one.
CM	Preserves comment lines of macro definitions; does not preserve comment lines that begin with ;; characters.Opposite option: NOCM.
CONTCK	Re-enables checksumming for instructions and data values; does not clear cumulative checksum.
DLD	Does not restrict DO-loop directives; suppresses error messages regarding directives in DO loops that do not make sense.Opposite option: NODLD, which is the default, as well as the reset setting at end of pass one.
INTR	Checks interrupt locations; lets the assembler screen interrupt vector locations of memory for inappropriate DSP instructions. Default setting, as well as the reset setting for absolute mode.Opposite option: NOINTR, which is the reset setting for relative mode.
LDB	Enables source listing debug: using the source listing instead of the assembly language file as the debug source file. Valid only if the command line includes the `-l` option.

Table continues on the next page...

Table 5-3. OPT Options (continued)

Option	Control Action
LPA	Enables hardware loop alignment for subsequent loops of the file; aligns loop addresses on a fetch set boundary, preventing additional pipeline stall cycles. The assembler implements this alignment automatically by padding execution sets with NOPs. However, this padding does not take place in hardware loops that do not use LOOPSTART notation.Opposite option: NOLPA, which is the default.
MB	Specifies the big memory model: move instruction addresses use the {a32} format instead of the default {a16} format.
NOAEC	Does not allow C escape characters. Default setting.
NOCC	Disables cycle counts, does not clear the total cycle count. Default setting, as well as the reset setting at end of pass one.Opposite option: CC.
NOCK	Disables checksumming. Default setting, as well as the reset setting at end of pass one.Opposite option: CK.
NOCM	Does not preserve comment lines of macro definitions.Opposite option: CM.
NODLD	Restricts DO-loop directives. Default setting, as well as the reset setting at end of pass one.Opposite option: DLD.
NOINTR	Does not check interrupt locations. Reset setting for relative mode.Opposite option: INTR.
NOLPA	Disables hardware loop alignment. Default setting.Opposite option: LPA.
NOOVLDBG	For files containing an overlay section, renames debug sections `.debug_info<overlay_scn>.` (In this format, multiple overlay sections in the same module corrupt the debug information.)Opposite option: OVLDBG, which is the default.
NOPS	Does not pack strings; stores one string byte per word.Opposite option: PS.
OVLDBG	Tells assembler to generate overlay-section debug information using local addresses instead of run addresses. This mode permits debugging of modules that contain multiple overlay sections. Default setting.Opposite option: NOOVLDBG.
PS	Packs strings for the DC directive; packs individual string bytes into consecutive target words.Opposite option: NOPS.
RSG	Specifies operations of both RSG_CHK and RSG_NOP options.
RSG_CHK	Specifies that: • DECEQA accepts SP as an operand, • EXTRACT does not check the 40-bit limit, • CMPEQA, CMPHIA, INCA, and DECA accept the same register twice as an operand.
SVO	Preserves object file if errors: overrides default setting to delete object files in case of errors. Must appear before any code or data generation.
RSG_NOP	Specifies:

Table 5-3. OPT Options

Option	Control Action
	• Combining all NOPs into one NOP, in the VLES prefix, • For FALIGN, introducing a separate VLES instead of combining NOPs in previous packets.

Example

```
OPT CEX,MEX    ; Enable DC and macro expansions
OPT CRE,MU       ; Print cross reference, memory utilization
```

5.3.61 ORG Initialize Memory Space and Location Counters

Sets absolute addresses; sets the memory space; and specifies and initializes the location counter.

```
ORG P[{lc|(ce)}]:[initval]
```

Parameters

`P`

P memory specifier.

`lc`

Location counter specifier: L (counter 1) or H (counter 2). Omitting both values specifies the default counter (counter 0) .

Counters are useful for providing mnemonic links among individual memory blocks.

`(ce)`

The counter number: a non-negative absolute integer expression. Must be in parentheses, must not exceed 65535.

`initval`

Optional initial value for the location counter. If you omit this value, the assembler uses the last value of the counter.

Remarks

Counters provide mnemonic links among individual memory blocks. The H, L, and default counter names are symbolic; the assembler does not verify that the H-counter value is greater than the L-counter value.

You may not use a label with this directive.

Examples

This first example:

```
ORG P:$1000
```

Sets the memory space to P, selects the P-space default counter, and initializes that counter to $1000.

This second example:

```
ORG PH:
```

Sets the memory space to P and selects the P-space H location counter (counter 2). This example does not initialize the counter, so the assembler uses the last H-counter value.

5.3.62 PAGE Advance Page or Size Page

Without argument values, advances the source listing to the top of the next page. With argument values, sets the size and margins of source-listing pages.

```
PAGE [pagewidth[,pagelength,blanktop,blankbtm,blankleft]]
```

Parameters

`pagewidth`

Number of columns per line: 1 through 255. The default is 80.

`pagelength`

Number of lines per page: 10 through 255. The default is 66. The special value 0 turns off all headers, titles, subtitles, and page breaks.

`blanktop`

Number of blank lines at the top of the page. The minimum (and default) value is 0. The value must maintain the relationship: `blanktop+blankbtm £ pagelength-10`.

`blankbtm`

Number of blank lines at the bottom of the page. The minimum (and default) value is 0. The value must maintain the relationship: `blanktop+blankbtm £ pagelength-10`.

`blankleft`

Number of blank columns at the left of the page; must be less than the `pagewidth` value. The minimum (and default) value is 0.

Remarks

If this directive has no arguments, the assembler does not print the directive in the source listing. If this directive does have argument values, the assembler includes the directive in the source listing.

Arguments may be any non-negative absolute integer expressions; commas without spaces must separate multiple argument values. Two adjacent commas tell the assembler to use the default value (or the last set value).

You may not use a label with this directive.

Examples

This first example:

```
PAGE 132,,2,4
```

Sets the page width to 132 columns, page length to 66 lines (the default), top margin to 2 lines, and bottom margin to 4 lines. This example specifies 0 blank columns (the default) at the left side of the page.

This second example given below merely advances the listing to the top of the next page:

```
PAGE
```

5.3.63 PMACRO Purge Macro Definition

Purges the specified macro definitions from the macro table, reclaiming space from the table. You may not use a label with this directive.

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

Parameter

`symbol`

A macro name. Commas without spaces must separate multiple `symbol` values.

Related Directive

MACRO

Example

```
PMACRO MAC1,MAC2
```

5.3.64 PRCTL Send Control String to Printer

Concatenates its arguments and sends them to the source listing, provided that the command line included the `-l` option.

```
PRCTL {exp | string},...
```

Parameters

`exp`

A byte expression that encodes non-printing control characters, such as ESC.

`string`

Any valid assembler control string.

Remarks

You may use this directive anywhere in the source file; the assembler outputs the control string at the corresponding place in the source listing. If a PRCTL directive is the first line of the first input file, the assembler outputs the control string before outputting page headings or titles.

If a PRCTL directive is the last line of the last input file, the assembler makes sure that all error summaries, symbol tables, and cross-references have been printed before it prints the control string. In this manner, you can use a PRCTL directive to restore the previous printer mode once printing is done.

If the command line did not include the -l option, the assembler ignores this directive. The assembler does not print this directive unless there is an error.

The assembler does not allow a label with this directive.

Example

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

5.3.65 SECFLAGS Set ELF Section Flags

Sets flag bits for the current section.

```
SECFLAGS flag[,flag...]
```

Parameter

`flag`

Any of these attribute values:

`write` - section contains writable data when loaded

`alloc` - section occupies memory space when loaded

`execinstr` - section contains executable instructions

`nowrite` - section contains non-writable data when loaded

`noalloc` - section does not occupy memory space when loaded

`noexecinstr` - section does not contain executable instructions

Remarks

Commas without spaces must separate multiple `flag` arguments.

Conventional sections (.text, .data, .rodata, or .bss) have specific type and flag values. A section with any other name is a code section; its default type and flag values are those of a .text section. But you can use the SECFLAGS and SECTYPE directives to redefine the default values.

You may not use a label with this directive.

Related Directives

SECTYPE, SECTION

Example

The SECTION directive begins a data section that has a non-standard name. Accordingly, the assembler gives this section the default flag values of a .text section: nowrite, alloc, execinstr. The SECFLAGS directive makes the flag values appropriate for a data section.

```
SECTION .data_input2
SECFLAGS write,alloc,noexecinstr
...
ENDSEC
```

5.3.66 SECTION Start Section

Starts a section: a block of relocatable code or data.

```
SECTION symbol [GLOBAL] [core_id]
.
.
section source statements
.
.
ENDSEC
```

Parameters

`symbol`

Name for the section. Standard names are `.text`, `.data`, `.rodata`, and `.bss`. Other names automatically invoke .text type and attribute values; other names must not duplicate a reserved name.

`GLOBAL`

Optional qualifier that makes all symbols defined within the section global. Without this parameter, all symbols defined within the section are local.

`core_id`

Optional 8102 DSP core loading destination for the section. Does not pertain to other processors.

Remarks

Code or data inside a section is independently relocatable within the memory space to which it is bound.

You can nest sections to any level. When the assembler encounters a nested section, it stacks the parent section and uses the nested section. Upon encountering the nested section's ENDSEC directive, the assembler restores and uses the parent section. (The ENDSEC directive always pairs with the closest previous SECTION directive.)

You can split a section into separate parts, by using the same name for multiple SECTION and ENDSEC pairs. This lets you arrange program source statements arbitrarily. For example, you can group all statements that reserve P space storage locations.

The assembler allocates a P-memory-space location counter for every section you define in source code. This counter maintains data and instruction offsets from the beginning of the section. At link time, according to your specifications, the linker relocates sections to an absolute address, loads sections in a particular order, or linked sections contiguously. The linker logically recombines split sections, making it possible to relocate each section as a unit.

You may give a section any name, except for the reserved names appearing in the below table.

Table 5-4. Reserved Section Names

.debug_abbrev	.debug_pubname	.rel.line
.debug_aranges	.default	.rel.line.debug_info
.debug_info	.line	.shstrtab
.debug_line	.mw_info	.strtab
.debug_loc	.note	.symtab
.debug_macro	.rel.debug_loc	

The table below explains the standard sections.

Table 5-5. Conventional ELF Sections

Section	Contents	Type	Attributes
.bss	Unititialized data	NOBITS	ALLOC, WRITE
.data	Initialized data	PROGBITS	ALLOC, WRITE
.mw_info	Assembler-generated contents that the linker consumes during dead data stripping	SHT_MW_INFO (SHT_LOPROC+3)	no sh_flags(0)
.note	User comments, as ABI 2.0 defines.	SHT_NOTE(7)	no sh_flags(0)
.rodata	Read-only initialized data	PROGBITS	ALLOC
.text	Program code	PROGBITS	ALLOC, EXECINSTR

If you do not use a standard name, the assembler assigns the .text type and attributes. To change these values, use the SECTYPE or SECFLAGS directives.

Symbols defined outside any section are global: they can satisfy an outstanding reference in the current file at assembly time, or in any file at link time. You may reference global symbols freely from inside or outside any section, as long as the global symbol name is unique.

Symbols defined within a section are local: they can satisfy an outstanding reference only within that section. But you can change this default arrangement:

- The GLOBAL qualifier of the SECTION directive makes all symbols defined in that section global symbols.
- The GLOBAL directive makes specified symbols global.

For the 8102 DSP, the linker can generate four linked core files, one for each processor, one of which contains the L2 memory. You must specify the core file into which the system will load the section. One way is to include a `core_id` specifier in the SECTION directive. The other method is to use the linker command file.

You man not use a label with this directive.

Related Directives

GLOBAL, ORG

Example

This directive starts a new section, TABLES:

```
SECTION TABLES GLOBAL
```

As the section has a non-standard name, its type and attributes are those of a .text section. The GLOBAL specifier means that all symbol definitions in the section define global symbols.

5.3.67 SECTYPE Set ELF Section Type

Defines the section type.

```
SECTYPE {progbits | nobits | overlay}
```

Parameters

`progbits`

Specifier for a section that has program contents, including code and data.

`nobits`

Specifier for a section that has no contents and does not occupy file space. (The assembler discards anything in `nobits` sections.)

`overlay`

Specifier for an overlay section.

Remarks

Standard sections `.text`, `.data`, `.rodata`, and `.bss` have default type and flag values; a section that has any other name receives the `.text` default values. Use this directive to change the default type value.

You may not use a label with this directive.

Related Directives

SECFLAGS, SECTION

Example

The SECTION directive starts new section `.data_output`. The non-standard name means that the section receives `.text` default type and attributes: PROGBITS, ALLOC, and EXECINSTR. The SECFLAGS directive changes the attributes; the SECTYPE directive changes the type to NOBITS.

```
SECTION .data_output
SECFLAGS write,alloc,noexecinstr
SECTYPE nobits
...
ENDSEC
```

5.3.68 SET Set Symbol to Value

Assigns the specified value to the specified symbol.

```
label     SET value
          SET label value
```

Parameters

`label`

A symbol.

`value`

An absolute expression. Must not contain any forward references.

Remarks

If you use this directive to define a label, another SET directive elsewhere in the program can redefine the label. Use this directive to establish temporary or reusable counters within a macro.

Related Directives

DEFINE, EQU, GSET

Example

```
COUNT   SET 0         ; Initialize count
```

5.3.69 SIZE Set Symbol Size

Sets the size of the specified symbol to the value of the expression parameter. May be anywhere in the source file, unless the symbol is a function name. If you define an INITIALIZER or VARIABLE symbol, the SIZE directive should appear after the symbol definition.

```
SIZE symbol,expression[,alignment]
```

Parameters

`symbol`

Any valid symbol. If a function name, the function definition must precede the SIZE directive.

`expression`

Any valid expression.

`alignment`

Optional alignment value for the symbol in the `.mw_info` section. (The linker uses this section's information for dead stripping.) This value must *not* be greater than the alignment of the symbol definition. This value *must* conform to the alignment of the symbol address.

Related Directive

TYPE

Example

```
_main:
.
.
RTS
SIZE _main,(*-_main)
```

5.3.70 STITLE Initialize Program Subtitle

Makes the specified string a subtitle of the program.

```
STITLE [string]
```

Parameter

`string`

Optional string value.

Remarks

The default program subtitle at the top of source-listing pages is blank. This directive specifies the subtitle for subsequent pages of the source listing. A subsequent STITLE directive changes the subtitle again. An STITLE directive without any string argument makes the subtitle blank.

The source listing does not include this directive.

You may not use a label with this directive.

Related Directive

TITLE

Example

```
STITLE 'COLLECT SAMPLES'
```

5.3.71 TITLE Initialize Program Title

Makes the specified string the title of the program.

```
TITLE [string]
```

Parameter

`string`

Optional string value.

Remarks

The default program title at the top of source-listing pages is blank. This directive specifies the title for subsequent pages of the source listing. A subsequent TITLE directive changes the title again. A TITLE directive without any string argument makes the title blank.

The source listing does not include this directive.

You may not use a label with this directive.

Related Directive

STITLE

Example

```
TITLE 'FIR FILTER'
```

5.3.72 TYPE Set Symbol Type

Sets the type for the specified symbol.

```
Label    TYPE typeid
```

Parameters

`Label`

Any label symbol of the program

`typeid`

Any of these values:

`FILE` - for the file name of the compilation unit.

`FUNC` - for a symbol associated with a function or other executable code.

`INITIALIZER` - for a symbol associated with an initializer.

`OBJECT` - for a symbol is associated with a variable, array, structure, or other such object.

`VARIABLE` - for a symbol associated with a variable.

Remarks

The assembler stores INITIALIZER- and VARIABLE- type information in the .mw_info section. The linker uses this information for data dead stripping. The linker stripping support document gives additional information about these initializers and variables.

Related Directive

SIZE

Example

```
Afunc   TYPE FUNC        ; Symbol Afunc is type STT_FUNC
```

5.3.73 UNDEF Undefine DEFINE Symbol

Cancels the substitution string for the specified DEFINE symbol.

```
UNDEF symbol
```

Parameter

`symbol`

Any symbol that a previous DEFINE directive specified.

Remarks

A previous DEFINE directive specified a substitution string for the symbol. This directive releases that substitution string; symbol no longer represents a valid DEFINE substitution.

You may not use a label with this directive.

Related Directive

DEFINE

Example

```
UNDEF   DEBUG    ; Undefine the debug substitution string
```

5.3.74 WARN Issue Programmer Warning

Outputs a warning, incrementing the warning count.

```
WARN {str | exp}[,{str | exp},...]
```

Parameters

`str`

Any valid string appropriate as part of a warning. Commas without spaces must separate multiple `str` values.

`exp`

Any expression appropriate as part of a warning. Commas without spaces must separate multiple `exp` values.

Remarks

Use this directive with conditional assembly directives for exceptional condition checking. The assembly proceeds normally after the assembler displays the warning.

You may not use a label with this directive.

Related Directives

FAIL, MSG

Example

```
WARN 'Parameter value too large'
```

5.4 Pragmas

There are two assembler pragmas, that are explained in this topic.

To use a pragma, follow this usage pattern:

```
PRAGMA <pragma_name> <parameters_list>
```

5.4.1 SECTYPE

Directs all tools in the compilation chain to consider all sections containing this pragma as siblings of `.init_table` sections.

```
PRAGMA sectype init_table
```

Parameter

`init_table`

Section type specifier.

Example

```
SECTION my_sec
PRAGMA sectype init_table
...
ENDSEC
```

5.4.2 STACK_EFFECT

Tells the assembler to propagate the specified function's stack effect to the linker.

```
PRAGMA stack_effect <symbol_name>,<size>
```

Parameters

`symbol_name`

Name of the function for which the assembler propagates stack-effect information.

`size`

Expression that represents the stack effect.

Remarks

For compiler-generated code, the compiler computes the stack effect. But you must specify the stack effect of assembler functions. When you provide this information to the linker, the linker can output the maximum stack effect of the function, in the mapfile.

Example

smth EQU 32 SECTION my_sec GLOBAL my_func my_func TYPE func PRAGMA stack_effect my_func,#smth move.l #smth,r3 adda r3,sp F_my_func_end EN DS EC

Chapter 6
Macros and Conditional Assembly

Macros streamline repeated patterns of code or groups of instructions. If you define such a pattern as a macro, you can call the macro at appropriate program locations instead of repeating the pattern.

For some patterns, variable values change for each pattern repetition. Other patterns involve conditional assembly. Macros accommodate either case: they let you designate selected statement fields as variable. You can call such a macro as many times as necessary, substituting different parameters for the variable fields.

In this chapter:

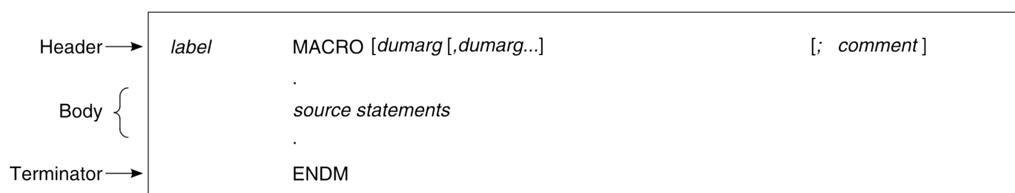
- Defining Macro
- Conditional Assembly

6.1 Defining Macro

Before you can use a macro, you must define it, either in the source file or in a macro library.

The below figure depicts a macro definition, which consists of these parts:

- Header - the MACRO directive, which assigns the name and defines dummy arguments.
- Body - the code and instructions the assembler uses for each macro call.
- Terminator - the ENDM directive.



The header, or MACRO directive, includes these parameters:

Figure 6-1. Macro Definition

The header, or MACRO directive, includes these parameters:

`label`

Name for the new macro. If this name duplicates any existing assembler directives or mnemonic opcodes, the assembler substitutes the macro for the directive or mnemonic opcode and issues a warning. This replacement does not happen if the label value duplicates a library macro name.

`dumarg`

Optional symbolic name to be replaced by an argument value when a statement calls the macro. Each `dumarg` value must follow the rules for global symbol names, and must not begin with the % character. Commas without spaces must separate multiple `dumarg` values.

`comment`

An optional comment.

Note that a macro definition can call other macros. Such other macros are *nested*. The definition of a nested macro must precede its appearance in a source-statement operation field. The assembler does not process calls or definitions of nested macros until it expands the parent macro.

6.1.1 Calling a Macro

Use a macro call to invoke a macro. In response, the assembler produces in-line code from the macro's statements, then inserts that code in the normal flow of the program. For every macro call, execution of the generated source statements takes place with execution of the rest of the program.

A macro call is a source statement that has this format:

```
[label]  macro  [arg[,arg...]]        [; comment]
```

where:

`label`

An optional label that corresponds to the location-counter value at the start of the macro expansion.

`macro`

The name of the macro; must be in the operation field.

`arg`

An optional, substitutable argument. Commas without spaces must separate multiple arguments.

`comment`

An optional comment.

These rules apply to macro arguments:

- Arguments must correspond one-to-one with the dummy arguments of the macro definition. If the call does not have the same number of arguments as the definition, the assembler issues a warning.
- Arguments can be quoted strings, although the assembler does not require single quotes around macro argument strings. However, single quotes must surround any embedded comma or space in an argument string.
- To declare a null argument for a macro call enter two commas without any intervening spaces; declare a null string for the argument; or terminate the argument list with a comma, omitting the rest of the argument list.
- The assembler does not substitute any characters in generated statements that reference a null argument.

6.1.2 Macro Expansions

The assembler generates source statements in response to a macro call. These source statements are a *macro expansion*. Macro expansions may contain substitutable arguments, and their types are relatively unrestricted. They can include any processor instruction, almost any assembler directive, or any previously-defined macro. Macro-expansion source statements must conform to the same conditions and restrictions that apply to statements that a programmer writes.

6.1.3 Macro Libraries

Macro libraries are directories of macro definitions. Each definition must be in a separate file, each file has the macro's name with the extension `.asm`. For example, `blockmv.asm` is the file that contains the definition of macro `blockmv`.

The MACLIB directive specifies a macro library directory. Upon encountering an unknown directive, the assembler first searches for the definition in the directive and mnemonic tables. If source code includes a MACLIB directive, the assembler also searches the specified directory. If the assembler finds the `.asm` definition file in the directory, it saves the current source line and the opens the file for input as an INCLUDE file. At the end of the file, the assembler restores the source line and resumes processing.

If the processed file does not include a macro definition of the unknown directive, the assembler issues an error message upon restoring and processing the source line. However, the processed file is not limited to macro definitions: it can include any valid source code statements. If multiple MACLIB directives specify several directories, the assembler searches the directories in their order in the MACLIB directives.

6.1.4 Dummy Argument Operators

The below table lists the text operators that permit argument text substitution during macro expansions. You can use these operators in macro definitions to concatenate text, convert numbers, and handle strings.

Table 6-1. Macro Dummy Argument Operators

Operator	Action
\	Concatenates a macro dummy argument with adjacent alphanumeric characters.
?	For the sequence `?symbol`, substitutes a character string that represents the `symbol` decimal value.
%	Converts the sequence `%symbol` to a character string that represents the `symbol` hexadecimal value.
"	Lets you use macro arguments as literal strings.
^	Evaluates local labels at normal, not macro, scope.

6.1.5 Macro Directives

Directives DUP, DUPA, DUPC, and DUPF duplicate subsequent source lines. Each directive is a special macro that simultaneously defines and calls an unnamed macro. Source statements you define with any of these directives must follow the same rules as macro definitions. For the DUPA, DUPC, and DUPF directives, such source statements can include macro dummy operator characters.

6.2 Conditional Assembly

Through conditional assembly a comprehensive source program can cover many conditions.

For macros, arguments specify assembly conditions. For the DEFINE, SET, and EQU directives, symbol definitions specify assembly conditions. Built-in assembler functions can test many conditions of the assembly environment.

You also can use conditional directives within a macro definition to make sure argument values are in appropriate ranges at expansion time. In this way, your macros can be self-checking and can generate error messages to any appropriate level of detail.

Use the directive pair IF and ENDIF, with the optional ELSE directive, to delimit a section of program for conditional assembly. Use the following format:

```
IF condition
.
source statements
.
[ELSE]
.

source statements
.
ENDIF
```

The assembler follows these rules for conditional assembly:

- If condition = TRUE (non-zero) and there is no ELSE directive, it assembles lines between the IF and ENDIF directives.
- If condition = FALSE (zero) and there is no ELSE directive, it ignores lines between the IF and ENDIF directives.

- If condition = TRUE (non-zero) and there *is* an ELSE directive, it assembles lines between the IF and ELSE directives and ignores lines between the ELSE and ENDIF directives.
- If condition = FALSE (zero) and there *is* an ELSE directive, it ignores lines between the IF and ELSE directives and assembles lines between the ELSE and ENDIF directives.

Index

;; Start Unreported Comment 89
; Start Comment 89
? Substitute Macro Value 91
^ Override Macro Local Label 93
" Delimit Macro String 93
" Expand DEFINE Quoted String 94
[ ] Delimit Substring 95
[ ] Group Instructions 96
@ Start Function 95
* Substitute Location Counter 95
\ Concatenate Macro Argument 90
\ Continue Line 90
#< Force Immediate Short Addressing 98
#> Force Immediate Long Addressing 99
Use Immediate Addressing 98
% Substitute Macro Hex Value 92
++ Concatenate Strings 95
< Force Short Addressing 96
> Force Long Addressing 97

A

ABS Absolute Value 47
Absolute and Relative Expressions 39
ACS Arc Cosine 47
Adding Debug Information 20
Address Assignment 75
ALIGN Align Location Counter 99
ARG Macro Argument 48
ASN Arc Sine 48
Assembler 11
Assembler Directives 85
Assembler Processing 30
AT2 Arc Tangent 49
ATN Arc Tangent 49

B

BADDR Set Buffer Address 100
BIGENDIAN Endian Mode Check 50
BSB Allocate Bit-Reverse Buffer 101
BSC Allocate Constant Storage Block 102
BUFFER Start Buffer 102

C

Calling a Macro 154
CCC Cumulative Cycle Count 50
CEL Ceiling 50
Checking Programming Rules 25
CHK Instruction/Data Checksum 51
CNT Macro Argument Count 51
Code Examples 26
COH Hyperbolic Cosine 52
Command-Line Options 14
Comment Field 33
COMMENT Start Comment Lines 104
Conditional Assembly 157
Constants
Controlling Assembler Messages 21
COS Cosine 52
Counting the core stalls 23
CTR Location Counter Number 53
CVF Convert Integer to Floating Point 53
CVI Convert Floating Point to Integer 54
CVS Convert Memory Space 54

D

Data Analysis Limitations 28
Data Analysis Terms 28
DCB Define Constant Byte 106
DC Define Constant 104
DCL Define Constant Long 107
DCLL Define Constant Long Long 107
DEF Defined Symbol 55
DEFINE Define Substitution String 108
Defining a Macro 153
Defining Substitution Strings 22
Descriptions 88
Directive List 86
DS Define Storage 109
DSR Define Reverse-Carry Storage 109
Dummy Argument Operators 156
DUPA Duplicate Sequence with Arguments 111
DUPC Duplicate Sequence with Characters 112
DUP Duplicate Source Lines 110
DUPF Duplicate Sequence in Loop 113

E

ELSE Start Alternative Conditional Assembly 114
ENDBUF End Buffer 116
END End of Source Program 115
ENDIF End Conditional Assembly 116
ENDM End Macro Definition 117
ENDSEC End Section 117
EQU Equate Symbol to Value 118
EXITM Exit Macro 119
EXP Expression Check 55
Expression Memory Space Attributes 40
Expressions 39

F

FAIL Issue Programmer Error Message 119
FALIGN Align with Fetch-Set 120
FLD Shift and Mask 56
FLR Floor 56
FRC Convert Floating Point to Fractional 57
Functions 45

G

Generating an Object File 20
GLOBAL Declare Global Section Symbol 121
GSET Set Global Symbol to Value 121

H

HIMEM Set High Memory Bounds 122

I

IF Start Conditional Assembly 123
INCLUDE Include Secondary File 124
Initialization File 29
Internal Expression Representation 41
INT Integer Check 57
Introduction 11

L

L10 Log Base 10 58
Label Field 32
LCV Location Counter Value 58
LEN String Length 59
LFR Convert Floating Point to Long Fractional 59
LIST List Assembly 125
LNG Concatenate to Double Word 60
LOG Natural Logarithm 60
LOMEM Set Low Memory Bounds 125
LST LIST Directive Flag Value 61
LUN Convert Long Fractional to Floating Point 61

M

MACLIB Specify Macro Library 126
MAC Macro Definition 61
MACRO Define Macro 127
Macro Directives 157
Macro Expansions 155
Macro Libraries 156
Macros and Conditional Assembly 153
Macros and DEFINE Symbols within Sections 74
MAX Maximum Value 62
Method 1: Absolute Mode 82
Method 2: Relative Mode 83
MIN Minimum Value 62
MSG Issue Programmer Message 128
MSP Memory Space 63
MULTIDEF Allow Multiple Definitions 129
Multi-Programmer Environment Example 81
MXP Macro Expansion 63

N

Nested and Fragmented Sections 73
NOLIST Stop Assembly Listing 129
NOTE Include Note 130
Numeric Constants 41

O

Operand Field 33
Operation Field 32
Operator Precedence 44
Operators 42
OPT Set Assembler Options 130
ORG Initialize Memory Space and Location Counters 136
Overlay Example 78
Overlay Manager 77
Overlays 75

P

PAGE Advance Page or Size Page 137
PMACRO Purge Macro Definition 138
POS Position of Substring 63
POW Raise to a Power 64
Pragmas 149
PRCTL Send Control String to Printer 139

R

Reading Input from an Argument File 19
Redirecting the Source Listing 21
REL Relative Mode 65
RND Random Value 65
RVB Reverse Bits in Field 65

S

SCP Compare Strings 66
Searching Additional Directories 22
SECFLAGS Set ELF Section Flags 140
Section Names 72
Sections 71
Sections and Relocation 74
Sections and Symbols 73
SECTION Start Section 141
SECTYPE 150
SECTYPE Set ELF Section Type 143
SET Set Symbol to Value 144

SGN Return Sign 66
Significant Characters 85
SIN Sine 66
SIZE Set Symbol Size 145
SNH Hyperbolic Sine 67
Software Development Flow 11
Software Project Management 71
Source Listing 36
Source Listing Example 37
Source Statements 31
Specifying a Target Architecture 24
Specifying Endian Mode 24
SQT Square Root 67
STACK_EFFECT 150
StarCore Assembler 13
Starting the Assembler 13
STITLE Initialize Program Subtitle 146
String Constants 42
Strings 35
Symbol Labels 35
Symbol Names 34

T

TAN Tangent 68
TITLE Initialize Program Title 146
TNH Hyperbolic Tangent 68
TYPE Set Symbol Type 147

U

UNDEF Undefine DEFINE Symbol 148
UNF Convert Fractional to Floating Point 69
Using an Environment Variable 18
Using OPT Options on the Command Line 23

V

Variable Length Execution Sets 33

W

WARN Issue Programmer Warning 149

X

XPN Exponential Function 69

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Information in this document is provided solely to enable system and software implementers to use Freescale products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. Freescale reserves the right to make changes without further notice to any products herein.

Freescale makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters that may be provided in Freescale data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including "typicals," must be validated for each customer application by customer's technical experts. Freescale does not convey any license under its patent rights nor the rights of others. Freescale sells products pursuant to standard terms and conditions of sale, which can be found at the following address: freescale.com/SalesTermsandConditions.

Freescale, the Freescale logo, CodeWarrior, QorIQ, StarCore are trademarks of Freescale Semiconductor, Inc., Reg. U.S. Pat. & Tm. Off. QorIQ Qonverge is a trademark of Freescale Semiconductor, Inc. All other product or service names are the property of their respective owners. The Power Architecture and Power.org word marks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org.

© 2009–2015 Freescale Semiconductor, Inc.

Document Number CWSCASMREF
Revision 10.9.0, 06/2015