CodeWarrior Development Studio for StarCore 3900FP DSPs C/C++ Compiler Reference Manual

Document Number: CWSCCMPREF
Rev. 10.9.0, 10/2015

Contents

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

Chapter 1
Introduction

Chapter 2
Tasks

Chapter 3
Concepts

Chapter 4
References

Chapter 1
Introduction

The StarCore C/C++ compiler, in conjunction with the assembler and the linker, generates binaries for the StarCore family of digital signal processors.

The *StarCore C/C++ Compiler User Guide* explains how to operate this compiler. In addition, the document describes you how to optimize C/C++ source code to let the compiler take full advantage of the StarCore processor's advanced hardware.

In this chapter:

1.1 About this document

The *StarCore C/C++ Compiler User Guide* is written using a task-based authoring approach. The document consists of:

- tasks chapter that consists of the following tasks:
  - **Compiler Configuration Tasks**: Help you configure the compiler according to your application requirements.
  - **General Compiler Tasks**: Help you use the compiler during the general development process.
  - **Pragmas and Attributes Tasks**: Help you to use pragmas that let you provide specific additional information to the compiler. The pragmas and attributes you specify provide context-specific hints that can save the compiler unnecessary operations, further enhancing optimization.
- concepts chapter that consists of the following concepts:
  - **Compiler Configuration Concepts**: Present compiler concepts that you might need to comprehend to accomplish compiler configuration tasks.

- **General Compiler Concepts**: Present compiler concepts that you might need to comprehend to accomplish general compiler tasks.
- **Pragmas and Attributes Concepts**: Present concepts that you might need to comprehend to use pragmas and attributes in your application, and accomplish pragmas and attributes tasks.

- appendices (reference information), such as command-line options, pragmas, pre-defined macros, etc. to which you might need to refer to accomplish compiler tasks. Refer to the Table 1-1 for more information about appendices.

In addition, each compiler task and concept provides a cross-reference to:

- related tasks
- related concepts
- related references

The following table lists and describes each chapter and appendix in this user guide.

Table 1-1. Structure of the StarCore C/C++ compiler user guide

Chapter/Appendix name	Description
Tasks	Consists of step-by-step instructions that help you configure and use the compiler.
Concepts	Consists of compiler concepts that you might need to comprehend to accomplish compiler tasks.
Command-line options	Lists all supported command-line options.
Pragmas and attributes	Lists all supported pragmas and attributes.
Runtime libraries	Lists and explains the runtime libraries included with the compiler.
Calling conventions	Lists and explains the calling conventions that the compiler supports.
Predefined Macros	Lists the compiler's pre-defined macros.
Intrinsics supported in emulation and MEX library	Lists the intrinsics supported in the MEX-library

1.2 Accompanying documentation

The Documentation page describes the documentation included in this version of CodeWarrior Development Studio for StarCore 3900FP DSP Architectures. You access the **Documentation** page by:

- a shortcut link on the Desktop that the installer creates by default, or
- opening `START_HERE.html` page from *CWInstallDir*`\SC\Help` folder.

1.3 Known limitations

The StarCore C++ compiler has the following known limitation.

- The size of initialization data

  The StarCore C++ compiler can handle up to 160M entries of initialization data. That is 40M entries if the data is int or short, 20M entries if the data is long long or floating point. If all entries of the initialization data are int or char, the compiler can handle up to 160M initialization data. But if all entries of the initialization data are short, long long, or floating point, the compiler can only handle up to 80M initialization data.

  In case you need to have huge initialization data, try to initialize it inside an initialization function, and call the initialization function before using the initialization data. If there are certain patterns in the initialization data, this method works well. If the initialization value for all entries is 0, do not initialize it.

- The compiler can not handle the `long` paths, which are beyond < 260 (length of the `windows_path_size`). If you specify a long path that is beyond the length of the `windows_path_size`, then the compiler throws following error message:

  ```
  [SCC,1,1001,-1]:ERROR: Cannot open option file, Probably because of the long path names.
  ```

- The compiler accepts only absolute paths for building projects on the command-line interface.
- Application file support is limited to section renaming layout settings and support for self contained libraries and components.
- Assembly functions should obey the standard calling convention.
- Inline assembly statements should follow the gcc syntax.

Chapter 2
Tasks

This chapter consists of step-by-step instructions that help you configure and use the compiler during the general development process.

In this chapter:

2.1 Compiler configuration tasks

In this section:

2.1.1 How to enable support for SC3900FP compiler

In order to generate the SC3900FP compiler code (default), use the following command:

```
scc -arch b4860 | sc3900fp
```

2.1.2 How to disable hardware floating point support in SC3900FP compiler

By default, hardware floating point support is enabled on SC3900FP compiler. In order to disable the hardware floating point single precision arithmetic and use the software library, use the following command:

```
scc -arch b4860 | sc3900fp -soft-fp
```

NOTE

The compiler defines `_SOFTFPA_` to distinguish between software and hardware floating point arithmetic.

Related Concept

- Understanding floating point support in SC3900FP compiler

Related Tasks

- How to enable support for SC3900FP compiler

Related References

- Floating-point characteristics (float.h)
- Floating-Point library interface (fltmath.h)

2.1.3 How to enable fused multiply and add generation

In order to enable fused multiply and add generation, use the following command:

```
scc -arch b4860 | sc3900fp -fma
```

Fused multiply and add are generated only if hardware floating point support is enabled on the SC3900FP compiler.

Related Concept

- Understanding floating point support in SC3900FP compiler

Related Tasks

- How to enable support for SC3900FP compiler
- How to disable hardware floating point support in SC3900FP compiler

Related References

- Floating-point characteristics (float.h)
- Floating-Point library interface (fltmath.h)

2.1.4 How to set environment variables

The StarCore compiler uses two environment variables to locate binaries and its license file:

- `SC100_HOME`

  Points to the compiler root directory that contains the `bin`, `include`, `lib`, and `rtlib` subdirectories. Set or reset the `SC100_HOME` environment variable by using the `set` command. For example:

  ```
  set SC100_HOME=<path_to_root_directory>
  ```

  You can override the `SC100_HOME` environment during run-time using the command-line interface. For example:

  ```
  scc -env <path_to_root_directory>
  ```

  The compiler ignores the path that the `SC100_HOME` environment variable specifies.

- `LM_LICENSE_FILE`

  Points to the license file. Set or reset the `LM_LICENSE_FILE` environment variable by using the `set` command. For example:

  ```
  set LM_LICENSE_FILE=<path_to_license_file>
  ```

  You can override the `LM_LICENSE_FILE` environment variable during run-time using the command-line interface. For example:

  ```
  scc -use-license-file <path_to_license_file>
  ```

  The compiler ignores the license file that the `LM_LICENSE_FILE` environment variable specifies.

Related Concepts

- Understanding compiler environment
- Understanding compiler startup code

Related References

- Predefined Macros

2.1.5 How to create user-defined compiler startup code file

Follow these steps:

1. Create a copy of the original startup code file. For example:

   ```
   copy install-dir\src\rtlib\startup\arch_name\startup.asm userdefined_startup.asm
   ```

   The `userdefined_startup.asm` file is your new startup code file.
2. Modify the `userdefined_startup.asm` file per your requirements.
3. Assemble the `userdefined_startup.asm` file by using the `scasm` command-line tool. For example:

   ```
   scasm -b -l userdefined_startup.asm
   ```
4. Instruct the compiler to use user-defined startup code file by using the `-crt` option. For example:

   ```
   scc -arch sc3900fp -crt userdefined_startup.eln
   ```

   Add the `-be` option to compile in big-endian mode. By default, the compiler compiles in little-endian mode.

Now, the compiler will use the user-defined startup code file.

Notes

- The `install-dir` directory refers to the default location where StarCore compiler directories are stored.
- The original startup code file is architecture specific. The `install-dir\src\rtlib\startup\` directory contains sub-directories for each supported architecture. Select the startup code file based on the architecture you are using.
- The compiler might not initialize the variables in the `.bss` section of a user-defined startup code file.

Related Tasks

- How to set environment variables
- How to write application configuration file

Related Concepts

- Understanding compiler startup code

Related References

- Predefined Macros

2.1.6 How to specify memory model

Table 2-1 describes the command-line options that you use to specify a memory model.

Table 2-1. Memory Model Command-line Options

Command-line Option	Description
`none` (default option)	Small memory model; use when all static data fits into the lower 64KB of the address space; all addresses must be 16-bit addresses
`-mb`	Big memory model; use when code, static data, and runtime library require 64KB to 1MB of memory; no restrictions on the memory allocated to addresses
`-mb1`	Big memory model with far runtime library calls; variation of big memory model; use when code and static data require more than 64KB but less than 1MB of memory for all, except the runtime code; the runtime code can be more than 1MB away from the rest of the program
`-mh`	Huge memory model; use when the application requires more than 1MB of memory

Follow these steps to specify a memory model:

- Using the command-line interface, specify the appropriate option, i.e. `-mb`, `-mb1`, or `-mh`. For example:

```
scc -arch sc3900fp -mb -O3 -be -g -ge -o test.eld test.c
```

- Using the CodeWarrior IDE, follow these steps:
  a. Start the IDE.
  b. In the **CodeWarrior Projects** view, select the project for which you want to modify the build properties.
  c. Select **Project** > **Properties**.

     The **Properties for** <project> window appears. The left side of this window has a Properties list. This list shows the build properties that apply to the current project.

d. Expand the C/C++ **Build** property.
e. Select **Settings**.
f. Use the **Configuration** drop-down list to specify the launch configuration for which you want to modify the build properties.
g. Click the **Tool Settings** tab.

   The corresponding page comes forward.

h. From the list of tools on the **Tool Settings** page, select **StarCore Environment**.

   The corresponding page appears.

i. Use the **Memory Model** drop-down list to specify the desired memory model.
j. Click **Apply**.

The memory model is specified, as per the requirements.

Notes

- For maximum efficiency, it is recommended that you place the program data in the lower 64KB of the memory map, so that the compiler can use the small memory model.

Related Concepts

- Understanding memory models

Related References

- *StarCore SC3000 Linker User Guide*

2.1.7 How to write application configuration file

NOTE

For SC3900FP compiler, application file support is limited to section renaming and layout settings.

An application configuration file gives you extensive control of compiler options and settings without having to specify individual pragma options in the source files.

Follow these steps to write an application configuration file:

1. Create a new text file and specify the extension as `.appli`
2. Specify the `configuration` and `end configuration` tags. You add all entries in an application file within `configuration` and `end configuration` tags.

3. Add a `view`. A `view` lets you define multiple sets of configuration definitions and settings in a single application file. You can either enable the `view` in the application file itself, or specify the view name when instructing the compiler to use an application file. The following listing shows the syntax for defining a `view`.
Listing: Defining a view

```
view <view_name>:
<view_body>:

    <section_definitions>

    <section_settings>

    <variable_settings>

    <module_list>: ; A module name is the file without the extension,
                   ; for example, foo.c is module foo

      module <module_name> [

        <variable settings>

        <function_list>: ; A function name is a C function with an
                         ; underscore (_) prefix

          function <function_name> [

            <variable settings>

          ]

      ]

end view
```

   a. Configure the section definitions, indicated by the `<section_definitions>` parameter

      A section definition binds logical names to the physical segments that the linker uses. This makes it easier to redefine mapping. A section definition applies to the entire `view`.

      The following listing shows the syntax.

      Listing: Section definition syntax

```
<section_definitions>:
    section

        <section_list>

    end section

<section>:

    <section_type> = [ <binding_list> ]

<section_type>:

    data

    program
```

```
    bss

    rom

    init

<binding>:

    <logical_name> : <physical_name> <optional_qualifier>

<qualifier>:[

    overlay
```

The following listing shows an example of the section definition.

Listing: Section definition example

```
...
section

    data = [data1: "My_Data_Seg1" core="c0", data2: "My_Data_Seg2"
overlay, data3: "My_Data_Seg3" overlay]

    program = [Pgm1: My_Pgm_Seg1"]

end section

...
```

b. Configure the section settings, indicated by the `<section_settings>` parameter

Once you define a section, you can specify a setting for each context. If you do not specify any settings, the section inherits the settings of its parent.

The following listing shows the syntax.

Listing: Section setting syntax

```
<section_settings>:
    <section_type> = <logical_name>

<section_type>:

    data

    program

    bss

    rom

    init
```

The following listing shows an example of the section settings.

Listing: Section settings example

```
...
module "My_Module" [

    ...

    data = data2       /*Use data2 as data space for whole module*/
```

```
    program = Pgm1      /*Use Pgm1 as program space for whole module*/

    ...

    function _alpha [

        ...

        data = data3  /*Use data3 as data space for function _alpha,*/
                      /*overriding data = data2 */

        ...

    ]

]

...
```

c. Configure the variable settings, indicated by the `<variable_settings>` parameter

Variable settings can be defined at either the `view` or the `module` level.

The following listing shows the syntax.

Listing: Variable settings syntax

```
<variable_settings>:
    <placement_list>

<placement>:

    place ( <variable_name_list> ) in <logical_name>
```

The following listing shows an example of the variable settings.

Listing: Variable settings example

```
...
view My_View"

    section

        data = [data1: "My_Data_Seg1", data2: "My_Data_Seg2" overlay,
data3: "My_Data_Seg3" overlay]

        program = [Pgm1: My_Pgm_Seg1"]

    end section

    place ( _A, _B, _C) in data2    /*Allocate globals _A, _B, and*/
                                    /*_C to segment data2*/

    place ( _X, _Y) in data3        /*Allocate globals _X and _Y*/
                                    /*to segment data3*/

    module "file1" [

        data = data2

        program = Pgm1

        place (_AB, _CD) in data1   /*Allocate global/static _AB,*/

                                    /*_CD to segment data1*/

    ]
```

```
        end view
        ...
```

4. Instruct the compiler to use the new application file by specifying the `-ma` option. For example:

```
scc ... -ma appconfig_filename.appli
```

The application configuration file is created.

Related Tasks

- How to create user-defined compiler startup code file

Related References

- Command-line options

2.1.8 How to align data to have packed structures

Add `__attribute__((__packed__))` or `(__PACK__)` to each structure that you want packed. The listing below shows an example.

Listing: Example - Packed structures

```
$ cat test.c
#include <stdio.h>

typedef struct X

{

        char c;

        int  i;

}       X;

typedef struct __attribute__((__packed__)) XP

{

        char c;

        int  i;

}       XP;

int main()

{

        printf("sizeof(X) == %d\n", sizeof(X));

        printf("offsetof(X, i) == %d\n", offsetof(X, i));
```

```
        printf("sizeof(XP) == %d\n", sizeof(XP));
        printf("offsetof(XP, i) == %d\n", offsetof(XP, i));
}
$ scc test.c -O3 -be -mb -slld -arch sc3900fp
$ runsim -d 3900iss a.eld
sizeof(X) == 8
offsetof(X, i) == 4
sizeof(XP) == 5
offsetof(XP, i) == 1
```

Related Tasks

- How to allow compiler to generate hardware loops

Related References

- Pragmas and attributes

2.1.9 How to inline or prevent inlining of function

The StarCore compiler automatically performs function inlining depending upon the optimization settings and the function size.

Follow these steps to inline or prevent inlining of a function:

- Specify the `always_inline` attribute to force the compiler to inline a specific function. The following listing shows an example.
  Listing: Forcing the compiler to inline a function

```
int foo ()
__attribute__((always_inline))
{
  ...
}
```

- Specify the `inline` keyword or `inline` attribute to force the compiler to inline a specific function with heuristics (see Understanding function inlining). The following listing shows an example.
  Listing: Forcing the compiler to inline a function

```
inline int foo ()
{
```

```
  ...
}
int foo () __attribute__((inline)) { ... }
```

- Specify the `noinline` attribute to prevent inlining of a specific function. The following listing shows an example.
  Listing: Preventing inlining of a function

```
int foo ()
__attribute__ ((noinline))
{
  ...
}
```

Notes

- The optimizer must be enabled for function inlining
- If the global optimization is not enabled, the called and the calling functions must exist in the same source file.
- The called and calling functions must have the same optimization level.
- Function inlining must not to be confused with `inline` keyword. Former is compiler implicit and latter is user enforced. The user can enforce function-inlining using `inline` keyword and several other methods, and such enforced inlining works at `-O0` (optimization level 0) as well. However, the compiler implicit function-inlining is triggered only on `-O2` and higher.

Related Concepts

- Understanding function inlining

Related Tasks

- How to write application configuration file

Related References

- Pragmas and attributes
- Pragmas to control function inlining

2.1.10 How to enable or disable warnings reporting

Follow these steps:

- Enable the reporting by specifying the `-Wmwfe-<keyword>` option using the command-line interface. For example, to enable all warnings on compiler front end, run this command:

```
scc -Wmwfe-all
```

- Disable the reporting by specifying the `-Wnomwfe-<keyword>` option using the command-line interface. For example, to disable all warnings on compiler front end, run this command:

```
scc -Wnomwfe-all
```

- Use a warning index value by specifying the `-W<index>` option using the command-line interface. For example:

```
scc -W1401
```

Notes

- The `mwfe` in `-Wmwfe` represents the compiler front end component. StarCore supports warnings reporting on other components in a similar way. For example, for StarCore linker, use `-Wlnk`, and for StarCore assembler, use `-Wasm`. However, the keywords for all components differ.
- For a list of keywords that you can specify with the `-Wmwfe` option, see Front-end Warnings Reporting Keywords in Command-line Options Appendix.
- For a list of index values that you can specify with `-W` option, see Warning index values in Command-line Options Appendix.

Related References

- Command-line options

2.2 General compiler tasks

In this section:

- How to use Word40 and unsigned Word40 data types
- How to use fractional data types
- How to pass command-line options using text file
- How to inline an assembly language instruction in C program
- How to inline block of assembly language instruction in C program
- How to call assembly language function existing in different file
- How to allow compiler to generate hardware loops
- How to keep compiler-generated assembly files
- How to compile C++ source files
- How to disassemble the source code
- How to use emulation library
- How to use MEX-library in MATLAB® environment
- How to enable code reordering
- How to disable automatic vectorization

2.2.1 How to pre-process a C source code file

A pre-processed `c` file includes expansions for all macros and header files.

Follow these steps to pre-process a `c` file:

- Using the command-line interface, specify the `-c` option followed by the pre-processed output file name. For example:

  ```
  scc -arch sc3900fp -O3 -be -g -ge -o test.eld test.c -c -C outfput_filename
  ```

  The `output_filename` is the pre-processed `c` file.

- Using the CodeWarrior IDE, follow these steps:
  a. Right-click on the respective `c` file in the CodeWarrior Projects view
  b. Select **Preprocess** option from the context menu

The pre-processed file appears in the new Editor tab.

Related Tasks

- How to optimize C/C++ source code

2.2.2 How to build projects using the command-line interface

The StarCore compiler lets you build projects using the command-line interface. You can manually build each source file individually, or use a `makefile` (using GNU `Make` utility).

Follow these steps to manually build a project using the command-line interface:

1. Collate all source files that you wish to include in the project
2. Build each source file individually using the `-c` option. For example:

```
scc -mb -be -arch sc3900fp -c -O3 file1.c
scc -mb -be -arch sc3900fp -c -O3 file2.c
scc -mb -be -arch sc3900fp -c -O3 file3.c
```

   You get individual object file for each source file.

3. Link the object files together using the `-o` option, and specify the final project name with `.eln` extension. For example:

```
scc -mb -be -arch sc3900fp -O3 -o project.eld file1.eln file2.eln file3.eln
```

The project is built manually using the command-line interface.

Follow these steps to use a `makefile` to build a project using the command-line interface:

1. Create a new `makefile`
2. Specify all header files that you wish to include in the project using `HEADERS` keyword. For example:

```
HEADERS = config.h image.h io.h piclayer.h bitstream.h blayer.h constants.h mblayer.h
motion.h
```

3. Specify all source files that you wish to include in the project using `CRCS` keyword. For example:

```
CSRCS = config.c image.c io.c piclayer.c bitstream.c blayer.c main.c mblayer.c motion.c
dct.c
```

4. Specify all object files (files with .eln extension) using `COBJ` keyword. For example:

```
COBJ = config.eln image.eln io.eln piclayer.eln bitstream.eln blayer.eln main.eln
mblayer.eln motion.eln dct.eln
```

5. Specify the compiler shell command. For example:

```
SCC = scc -mb -be -arch sc3900fp
```

6. Create a rule so that each `.eln` file maps to correct source file. For example:

```
%.eln:%.c $(HEADERS)
$(SCC) -c $(COPTS) $*.c
```

7. Create a rule to link all the object files together. For example:

```
sencoder: $(HEADERS) $(SOBJ) Makefile
$(SCC) $(COPTS) -o sencoder $(SOBJ)
```

8. Specify the `clean` keyword to clean the intermediate files, and force a rebuild. For example:

```
clean:
/bin/rm -r -f *~ sencoder *.eld *.eln *.sl
```

9. Optionally, create a rule for "all" if you have multiple builds.

The project is built using `makefile`.

Example

The listing below shows an example of a `makefile` for the StarCore compiler.

Listing: Example makefile for StarCore compiler

```
HEADERS = config.h image.h io.h piclayer.h bitstream.h blayer.h \
          constants.h mblayer.h motion.h
COPTS = -O3

CSRCS = config.c image.c io.c piclayer.c bitstream.c blayer.c main.c \
        mblayer.c motion.c dct.c

COBJ = config.eln image.eln io.eln piclayer.eln bitstream.eln \
       blayer.eln main.eln mblayer.eln motion.eln dct.eln

SCC = scc -mb -be -arch sc3900fp

all: sencoder

 echo Update to date

%.eln:%.c $(HEADERS)

 $(SCC) -c $(COPTS) $*.c

sencoder: $(HEADERS) $(SOBJ) Makefile

 $(SCC) $(COPTS) -o sencoder $(SOBJ)

clean:

 /bin/rm -r -f *~ sencoder *.eld *.eln *.sl

...
```

The listing below shows the output of using `makefile` that is listed in the above listing.

Listing: Output of StarCore compiler makefile

```
$ make
scc -mb -be -arch sc3900fp  -c -O3 config.c

scc -mb -be -arch sc3900fp  -c -O3 image.c

scc -mb -be -arch sc3900fp  -c -O3 io.c

scc -mb -be -arch sc3900fp  -c -O3 piclayer.c

scc -mb -be -arch sc3900fp  -c -O3 bitstream.c

scc -mb -be -arch sc3900fp  -c -O3 blayer.c

scc -mb -be -arch sc3900fp  -c -O3 main.c

scc -mb -be -arch sc3900fp  -c -O3 mblayer.c

scc -mb -be -arch sc3900fp  -c -O3 motion.c

scc -mb -be -arch sc3900fp  -c -O3 dct.c
```

```
scc -mb -be -arch sc3900fp  -O3 -o sencoder config.eln image.eln io.eln
piclayer.eln bitstream.eln blayer.eln main.eln mblayer.eln motion.eln
dct.eln

echo Update to date

Update to date
```

Notes

- You can use `Cygwin` tool for `Make` utility.

Related Tasks

- How to pass command-line options using text file

Related Concepts

- Understanding compiler environment

Related References

- Command-line options

2.2.3 How to improve performance of the generated code

In order to improve performance of the generated code, provide extensive details about the source code, data, and target architecture to the compiler.

Follow these techniques to improve performance of the generated code:

- Enable source code optimizations. See How to optimize C/C++ source code.
- Select the best data types, including intrinsic functions if applicable. See How to use fractional data types.
- Signed loop counters are to be preferred to unsigned ones. To obtain the best optimizations, please use signed counters whenever possible.
- It is preferred to align the data. The compiler speculates SC3900FP features and generates vector load/store instructions for unaligned data, but note that this feature is not available on the SC3900FP architecture for non-cacheable data. The SC3900FP compiler must be informed, by disabling vectorization. See #pragma novector and __attribute__((novector)).
- Be cautious about pointer aliasing, use the `restrict` keyword where applicable. See How to use restrict keyword.
- Make sure that the source code is alias by type compliant. As per the alias by type rules, a value which is stored in memory should always be accessed using the same

access size. For more information, refer Chapter 6.5, Paragraph 7 in C99 Standard document.

For example, the following source code is not alias by type compliant.

```
short* x;
y = *x;
z = *(long*)x;
```

In this example, x is accessed once as 2 bytes and once as 4 bytes. If you want to ignore alias by type rules, use #pragma alias_by_type or command line option `-opt=[no]alias_by_type` in Optimization and code options. However, some optimization opportunities may be missed.

- Make sure that the compiler can generate hardware loops. See How to allow compiler to generate hardware loops.
- Make sure to use the likely and unlikely macros where possible. See How to enable code reordering.

Related Concepts

- Understanding optimizer
- Understanding intrinsic functions
- Understanding code reordering

Related References

- Command-line options

2.2.4 How to improve compilation speed

In order to improve compilation speed, follow these techniques:

- Disable optimizations. For example, you can enable optimization for size rather than for speed. This will usually disable some time consuming optimizations in the compiler, and improve the compilation speed. You can also disable those optimizations that help in improving performance of the generated code. The extreme case is to disable all optimizations.

  Using the command-line interface,

  - to disable optimization for speed, remove the `-O3` option
  - to disable global optimization, remove the `-Og` option
  - to disable optimization for size, remove the `-Os` option

- to enable optimization for size, add the `-Os` option
- to disable all optimization, add the `-O0` option, or remove all the optimization flags.

Using the CodeWarrior IDE, follow these steps:

a. Start the IDE.
b. In the **CodeWarrior Projects** view, select the project for which you want to modify the build properties.
c. Select **Project > Properties**.

   The **Properties for <project>** window appears. The left side of this window has a Properties list. This list shows the build properties that apply to the current project.

d. Expand the C/C++ **Build** property.
e. Select **Settings**.
f. Use the **Configuration** drop-down list to specify the launch configuration for which you want to modify the build properties.
g. Click the **Tool Settings** tab.

   The corresponding page comes forward.

h. From the list of tools on the **Tool Settings** page, select **Optimization** under **StarCore C/C++ Compiler**.

   The corresponding page appears.

i. Use the **Optimization Level** drop-down list to specify the desired optimization level. You can also enable/disable **Global Optimization** and select **Smart Unrolling**.
j. Click **Apply**.

For more advanced control, you can pass individual optimization flags in the application configuration file. See How to write application configuration file for more information.

NOTE

- Do not include unnecessary header files. This will reduce unnecessary recompilation due to header file change. Break the header file if necessary.
- Break large function into smaller functions. Smaller functions usually take less time to compile.
- Avoid large initialization data. Large initialization data resides in the memory and is likely to slow down compilation.

Related Concepts

- Understanding optimizer
- Understanding intrinsic functions

Related References

- Command-line options

2.2.5 How to optimize C/C++ source code

Follow these steps:

1. Enable the optimizer by specifying an optimization level.
   - Using the command-line interface, specify the `-O0, -O1, -O2, -O3, or -O4` option. For example:

     ```
     scc -O3 -o test.eld -arch sc3900fp test.c
     ```

   - Using the CodeWarrior IDE, follow these steps:
     1. Start the IDE.
     2. In the **CodeWarrior Projects** view, select the project for which you want to modify the build properties.
     3. Select **Project > Properties**.

        The **Properties for <project>** window appears. The left side of this window has a Properties list. This list shows the build properties that apply to the current project.

     4. Expand the C/C++ **Build** property.
     5. Select **Settings**.
     6. Use the **Configuration** drop-down list to specify the launch configuration for which you want to modify the build properties.
     7. Click the **Tool Settings** tab.

        The corresponding page comes forward.

     8. From the list of tools on the **Tool Settings** page, select **Optimization** under **StarCore C/C++ Compiler**.

        The corresponding page appears.

     9. Use the **Optimization Level** drop-down list to specify the desired optimization level.
     10. Click **Apply**.

- To set a specific optimization level at a file level, specify the `opt_level` pragma at the beginning of the file. However, you can place the `opt_level` pragma anywhere in the file. The optimization level that the `opt_level` pragma specifies takes precedence over the optimization level specified at the project level, or at the command-line interface. The listing below shows an example of the `opt_level` pragma used at a file level.
  Listing: Setting optimization level at file level

```
/* foo.c */
/* sets the opt level for foo.c - applicable to both foo() and bar()*/


#pragma opt_level = "O3"


int foo ()

{

...

}

void bar ();

{

...

}
```

- To set a specific optimization level at a function level, specify the opt_level pragma immediately after the opening brace of the function. The optimization level that the `opt_level` pragma specifies takes precedence over the optimization level specified at the project level, or at the command-line interface. The listing below shows an example of the `opt_level` pragma used at a function level.
  Listing: Setting optimization level at function level

```
int foo ()
{

  #pragma opt_level = "O3"

  ...

}
```

Notes

- You can apply specific levels of optimizations at the project, file, or function level. However, to apply a specific level of optimization at the file or function level, use the `opt_level` pragma, or the application configuration file.

Related Tasks

- How to write application configuration file
- How to pass loop information to compiler optimizer

Related Concepts

- Understanding optimizer

Related References

- Command-line options
- Pragmas and attributes

2.2.6 How to pass loop information to compiler optimizer

The loop information, such as minimum number of iterations, maximum number of iterations, and modulo and remainder, help the compiler optimizer to perform loop unrolling. The loop unroll process replicates the loop body and executes multiple iterations at once, enhancing the performance.

Use the `loop_count` pragma to pass loop information to compiler optimizer. You can specify these details using the `loop_count` pragma:

- Minimum number of iterations of the loop. When specified as non-zero, the compiler can remove the code that checks for zero loop count, enhancing the performance.
- Maximum number of iterations of the loop.
- Modulo and remainder. Used for loop unrolling information. If the loop always executes a multiple of 2 or 4 times, the compiler can use this information to unroll the loop correctly.

For example:

```
for(j=0; j<refSize; j+=2) {
  #pragma loop_count (4,40,2,0)
... }
```

In this example,

- the loop executes a minimum of 4 times
- the loop executes a maximum of 40 times
- the loop always iterates with a multiple of 2
- the last 2 items represent the modulo property of the trip-count and the remainder

Notes

- The `loop_count` pragma is always placed immediately after the opening brace of the loop.

- It is possible to set a modulo and remainder. The specified example uses a modulo of 2 and a remainder.
- The `loop_count` pragma can enable the compiler for other optimizations as well.

Related Tasks

- How to optimize C/C++ source code

Related Concepts

- Understanding optimizer

Related References

- Pragmas and attributes

2.2.7 How to use restrict keyword

The `restrict` keyword instructs the compiler that a pointer does not alias with another pointer. A pointer aliases with another pointer when both the pointers access the same object. The `restrict` keyword improves the compilation performance because the compiler can put the pointers' memory accesses in parallel.

Define the respective pointer, reference, or array with the `restrict` keyword. The following listing shows an example.

Listing: Using restrict keyword

```
void foo (short * restrict a, short * restrict b, int N)
  int i;

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

    b[i]=a[i]+2;

  }

  return;

}
```

Notes

- The `restrict` keyword is a type qualifier that can be applied to pointers, references, and arrays
- The `restrict` keyword must be used with care, when it is guaranteed that it cannot alias with another pointer in its scope. Otherwise, it can generate unexpected results, as the compiler does not re-validate the information provided by the application.

Related References

- Pragmas and attributes

2.2.8 How to use Word40 and unsigned Word40 data types

`Word40`/ `UWord40` is an additional non-standard 40-bit integer type between `long int` and `long long int`.

Starting with CW for StarCore v10.4.1, `__int40/unsigned __int40` is considered a basic integer native compiler type, instead of a structure.

The following operations are supported directly on `__int40/unsigned __int40` variables. Intrinsics are not needed to perform these operations.

- initialization
- compare: `>, >=, <, <=, ==, !=`
- arithmetic operations: `+,-,*,/,%,>>,<<,++,--`
- logic operations: `&, |, ^, !, ~, ||, &&`
- passed and returned in data registers

`__int40/unsigned __int40` are supported with the following limitations:

- `__int40/unsigned __int40` cannot be used for bitfields
- Formatted I/O should be done using `long long` (`%lld`, `%llx`), `unsigned long long` (`%llu`, `%llx`)

Listing: __int40/unsigned __int40 - Example

```
typedef __int40 int40;
typedef unsigned __int40 uint40;

int40 g1 = 0x1234567890;

int40 g2 = 0x1;

uint40 ug1 = 0xfffffffffd;

uint40 ug2 = 0x2;

g2 = (int40) ug2;

g1 = g1 + g2;

ug2 = ug1 * ug2;

printf("g1 == %lld; g2 == %llu\n", (long long)g1, (unsigned long
long)ug1);
```

2.2.9 How to use fractional data types

Fractional data types are not supported in native C language. However, the StarCore compiler provides fractional intrinsic functions to support fractional data types. The type of the operation performed, be it integer or fractional, is indicated by the intrinsic syntax. The absence of the "i" flag in the name of an intrinsic indicates a fractional operation.

For instance, for the multiply-accumulate operation which is different on integer than on fractional, `__mac_i_x_ll` indicates an integer multiply-accumulate, while `__mac_x_hh` refers to a fractional non-saturating multiply-accumulate and `__mac_s_x_hh` refers to a fractional saturating multiply-accumulate. It is important that appropriate fractional intrinsic are used for fractional operations.

The StarCore architecture supports both integer and fractional instructions. Fractional instructions perform an implicit left shift after the multiplication and saturate, whereas integer instructions do not.

Related Concepts

- Understanding fractional and integer arithmetic
- Understanding intrinsic functions

Related References

- Pragmas and attributes
- *StarCore C Compiler Intrinsics Reference Manual*

2.2.10 How to pass command-line options using text file

Using a text file to pass command-line options might be useful when:

- the length of the command-line text exceeds the OS imposed limit
- calling the compiler using a script that generates or modifies the command-line options

Specify the `-F` option followed by the text file name to pass command-line options using a text file. For example:

```
scc -F commandfile.txt
```

Notes

- You may specify the `-F` option multiple times in a single command. In such a case, the contents from all command files are concatenated, and then combined with any switches specified directly at the command-line.

Related Concepts

- Understanding compiler environment

Related References

- Command-line options

2.2.11 How to inline an assembly language instruction in C program

To inline a single assembly instruction, use the following syntax:

```
asm ("instruction_text" : output_operands : input_operands);
```

Here, `instruction_text`, is the mnemonic followed by a template for the operands. The numbers in the operand's template refer the operands in `output_operands` and `input_operands`. The first operand on the left is in index 0. The indices of `input_operands` are in continuation of those of the `output_operands`.

After allocating registers to the C variables, these registers are emitted in the assembly instruction according to the operands' template.

output_operands and input_operands specify the C variable operands in the instruction.

The syntax to specify an operand is:

```
"constraint" (variable name)
```

Here, `constraint` is a storage location class that may be preceded by `=` or `+`.

The available classes are:

- `d8, d16, d32, d40, d64` for a `d` register with the size specified.
- `r8, r16, r32` for a `r` register with the size specified.
- `f32, f64` for floating point.

It is important to specify size. If an operand uses `d40` as a constraint and the variable is of type `int`, then the variable would be sign extended before the assembly instruction.

Only one memory (`ml`/ `ms`) operand is allowed in an assembly statement.

- `ms` refers to a store memory operand
- `ml` refers to a load memory operand

The type of the memory operand must be of the same size as the memory accessed by the instruction.

- If a constraint is preceded by `=`, it means that this operand value is changed by the instruction.
- If a constraint is preceded by `+`, it means that this operand value is both used and changed by the instruction.

For example:

```
asm("subc.wo.leg.x %2, %1, %0": "=d40" (r): "d40" (__Db), "d40" (__Da));
```

In this example, the mnemonic `subc.wo.leg.x` is followed by a template for the operands, `%2`, `%1`, `%0`. `r` has index `0`, `__Db` has index `1`, and `__Da` has index `2`.

If `r` is in `d15`, `__Db` is in `d3`, and `__Da` is in `d13`. The assembly instruction that will be emitted is:

```
subc.wo.leg.x d13, d3, d1
```

`r` is a `d` register which is changed by the instruction. `__Db` and `__Da` are d registers used by the instruction.

Some more examples:

```
asm("macdrf.h.r.4t %4:%5, %2:%3, %0:%1": "+d40" (*__De), "+d40"
(*__Df): "d32" (__Dc), "d32" (__Dd), "d32" (__Da), "d32" (__Db));  /*
__De and __Df are both used and defined by this instruction */

asm("eor.l 0x80000000 %1, %0":  "=f32" (__Dm) : "f32" (__Da));
//   __Dm / __Da are float

asm("ld.par.w (%1), %0": "=d32" (*__Da) :  "ml"
(*__Ra));                                                          /* __Ra is
a pointer */

asm("st.2bf %1:%2, (%0)":  "=ms" (*__Ra) : "d32" (__Da), "d32" (__Db));
/* __Ra is a pointer */
```

Related Tasks

- How to inline block of assembly language instruction in C program
- How to call assembly language function existing in different file

Related References

- Command-line options

2.2.12 How to inline block of assembly language instruction in C program

Follow these steps to inline a block of assembly instructions in a `c` program:

1. Map every parameter to the standard calling convention.
2. Map the return value, if any, to the standard calling convention.
3. All registers which are used in the assembly instructions must be the one marked as modified at function call by the standard calling convention.
4. Old .arg and .reg constructs generate errors.
5. Enclose the block of assembly instructions within `asm_body` and `asm_end` tags.

The listing below shows an example of an inlined assembly function that takes two input parameters from registers `r0` and `r1`, then returns the result in `r0`.

Listing: Inlining a block of assembly instructions in a C program

```
asm int t6(int param1, int *param2)
{

asm_body

    ld.l (r1),r1

    adda.lin r0,r1,r0

asm_end

}
```

The listing below shows an example of labels and hardware loops within inlined assembly language function. You should use hardware loops within assembly language functions only if you know that the loop nesting is valid. In this example, the function is called from outside a loop, so hardware loops are allowed. Note that in order to inline an assembly function in a calling C function, you need to specify that all labels in the inline assembly function are considered as local labels. You append `%C` to a label's name to make it local.

Listing: Inlined assembly function with labels and hardware loops

```
#include <stdio.h>
char sample[10] = {9,6,7,1,0,5,1,8,2,6};

int status;

asm char t7(int p)

{

asm_body

    doen.3 r0

    eor.x d8,d8,d8 tfra.l #_sample,r1

skip.3 _L10%C
```

```
    loopstart3
    ld.b (r1),d1
    add.x d8,d1,d8
    add.x #1,d1,d1
    st.b d1,(r1)+
    loopend3
_L10%C:move.l d8,r0
asm_end
}
int main()
{
    int m = 8;
    int s,i;
    for(i=0;i < 10;i++) {
        sample[i] *= 2;
        printf("%d ",sample[i]);
    }
    printf("\n");
    s = (int)t7(m);
    printf("S= %d\n",s);
    for(i=0;i < 10;i++)
        printf("%d ",sample[i]);
    printf("\n");
    return 1;
}
```

The listing below shows an example of global variable references and an identifier list within an inlined assembly language function. To access global variables, you use their linkage name. The default linkage name is the variable name with an underscore (_) prefix. For example, to access variables `vector1` and `vector2` within the function, use linkage names `_vector1` and `_vector2`.

Listing: Referencing global variables in an inlined assembly function

```
asm void test(int n, short *r1)
{
asm_body
    tfra.l #_vector1,r6
    tfra.l #_vector2,r11
```

```
    tfra.l #_result_2,r7
    addla.1.lin r0,r6,r6
    addla.2.lin r0,r11,r11
    ld.w (r6),d0
    ash.rgt.x #<2,d0,d0
    st.w d0,(r1)
    ld.l (r11),d1
    ash.lft.x #1,d1,d2
    st.l d2,(r7)
asm_end
}
int main(void)
{
    test(12,&result_1);
    printf("Status = %d %d\n",(int)result_1, result_2);
    return (int)result_2;
}
```

When inlining a block of assembly language instruction in a C program, follow these guidelines in addition to the guidelines that you follow when inlining a single assembly language instruction:

- All inline assembly functions should obey the standard calling convention.
- Avoid statements, such as RTS. Even if you define a function as standalone, it is possible that the compiler includes this function in a sequence of instructions. In such a case, the compiler automatically adds the necessary return statements.
- If the assembly language function requires local variables, you must allocate them specifically on the stack, or define them as static variables. The compiler does not automatically allocate local variables for use by the assembly language functions.
- Assembly language functions you define as a block of instructions can access global variables in the C source code. This is because such variables are static by definition.

Related Tasks

- How to inline an assembly language instruction in C program
- How to call assembly language function existing in different file

Related References

- Command-line options

2.2.13 How to call assembly language function existing in different file

Follow these steps to call an assembly language function that exists in a different file than your C/C++ file:

1. Make sure the assembly function uses the standard calling conventions.
2. Assemble the file that contains the assembly function. However, you may skip this step.
3. Define the assembly function as an external function in the C/C++ program.
4. Specify the file that contains the assembly function as an additional input file when compiling the C program.
5. See the listing below for an example file that contains an assembly function.
   Listing: Example file containing an assembly function

```
;
; extern void fft(short *, short*);

;

; Parameters: pointer to input buffer in r0

; pointer to output buffer in r1

;

_fft:

;Save and restore d28-d31 and r28-r31, according to the default calling
conventions

    push.4x d28:d29:d30:d31

    push.4l r28:r29:r30:r31

    <implementation of FFT algorithm>

    pop.4l r28:r29:r30:r31

    pop.4x d28:d29:d30:dd31

rts
```

6. The listing below shows how you compile and run this .cpp file enabling the support for exception handling and double type. for an example C file that calls the assembly function.
   Listing: C file calling an external assembly function

```
#include <stdio.h>
extern void fft(short *, short*);

short in_block[512];

short out_block[512];
```

```
int in_block_length, out_block_length;

void main()

{

    int i;

    FILE *fp;

    int status;

    in_block_length=512;

    out_block_length=512;

    fp=fopen("in.dat","rb");

    if( fp== 0 )

    {

        printf("Can't open parameter file: input_file.dat\n");

        exit(-1);

    }

    printf("Processing function fft \n");

    while ((status=fread(in_block, sizeof(short), in_block_length, \
          fp)) == in_block_length)

    {

        fft(in_block,out_block);

    }

}
```

When using the C++ language, replace `extern void fft(short *, short*)` as shown in the above listing, how you compile and run this `.cpp` file enabling the support for exception handling and `double` type, with `extern "C" void fft(short *, short*)` to avoid function name mangling by the C++ language. The listing below shows an example.

Listing: C++ file calling an external assembly function

```
#include <stdio.h>
extern "C" void fft(short *, short*);

short in_block[512];

...

...
```

Notes

- To enable the compiler to add function debug information in the generated assembly file, the function must be enclosed within the *funcname* `type func` and the `F`*funcname*`_end` tags, where *funcname* represents the function name. For example:

```
_foo type func

ld.l (r1),r1

suba.lin r0,r1,r0

rts

F_foo_end
```

- The optimizer ignores inlined assembly instructions
- The compiler can find errors in the assembly instructions only when assembling the source files
- When inlining an assembly instruction, make sure that:
  - identifiers are prefixed with an _ (underscore)
  - registers are prefixed with a $ (dollar sign)
  - labels are suffixed with a . (period)

Related Tasks

- How to inline an assembly language instruction in C program
- How to inline block of assembly language instruction in C program

Related References

- Command-line options

2.2.14 How to allow compiler to generate hardware loops

The StarCore compiler supports four levels of hardware loops. A hardware loop:

- starts with a count equal to the number of iterations of the loop
- decrease by one on each iteration (uses a step size of -1)
- decreases when the loop counter is zero

The compiler can generate a hardware loop by interpreting a normal C/C++ language loop. Follow these guidelines:

- Avoid function calls in the loop

  A function call in the loop body prevents the compiler from generating a hardware loop. For example:

```
for(j=0; j<refSize; j+=2) {

int a=0;
```

```
a= foo (a); // might block hardware loop generation

}
```

- Keep loop conditions simple

  A dynamic or a complicated loop end count that cannot be resolved prevents the compiler from generating a hardware loop. For example:

```
for(j=0; j<N; j+=2) // simple

for(j=0; j<(ptr+4); j++) // complex, simplify
```

- Make sure that the loop step is 1, a power of 2, or is equal to 3, 5, or 7. For example:

```
for(j=0; j<N; j+=2) // OK

for(j=0; j<N; j+=3) // OK

for(j=0; j<N; j+=5) // OK

for(j=0; j<N; j+=20) //might block hardware loop
```

- Use `short` type for loop indices and bounds. For example,

```
void foo (short N) {

short j;

for(j=0; j<N; j+=2)
```

Notes

- The compiler can generate a hardware loop from a loop that contains function calls, if the calling convention guarantees that the callee does not modify the LC registers or if the callee is part of the same compilation unit as the caller (same file) and it does not modify the LC registers.

Related References

- Command-line options

2.2.15 How to keep compiler-generated assembly files

It is recommended that you frequently examine the compiler-generated assembly code to get information on how to further optimize the source code.

Follow these steps to keep the compiler-generated assembly files:

- Using the command-line interface, specify the `--keep` option. For example:

```
scc -O3 -o test.eld -arch sc3900fp --keep test.c
```

- Using the CodeWarrior IDE, follow these steps:
  a. Start the IDE.
  b. In the **CodeWarrior Projects** view, select the project for which you want to modify the build properties.
  c. Select **Project** > **Properties**.

     The **Properties for** <project> window appears. The left side of this window has a Properties list. This list shows the build properties that apply to the current project.

  d. Expand the C/C++ **Build** property.
  e. Select **Settings**.
  f. Use the **Configuration** drop-down list to specify the launch configuration for which you want to modify the build properties.
  g. Click the **Tool Settings** tab.

     The corresponding page comes forward.

  h. From the list of tools on the **Tool Settings** page, select **Output Listing** under **Compiler** .

     The corresponding page appears.

  i. Check the **Keep assembly (.sl) files** checkbox.
  j. Click **Apply**.

Example

In the listing below, the numbers in the bracket relate to the original source code. The first number represents the line number of original `C/C++` language statement.

Listing: Example compiler-generated assembly file

```
;*****************************************************************
;*                                                               *
;* Function Name:             _vec_ex                            *
;* Stack Frame Size:          0 (0 from back end)                *
;* Calling Convention:        17                                 *
;*                                                               *
;*****************************************************************

    GLOBAL    _vec_ex
    ALIGN     16
_vec_ex   TYPE    func    OPT_SPEED
    SIZE      _vec_ex,F_vec_ex_end-_vec_ex,16
;PRAGMA stack_effect _vec_ex,0
[
```

```
    tfra.l    #100,r0     ;[0,1]
    tfra.l    #_b,r11     ;[0,1]
    tfra.l    #_a,r9      ;[0,1]
]
[
    doen.2    r0          ;[0,1]
    tfra.l    #_c,r10     ;[0,1]
    tfra.l    #2,r0       ;[0,0]
]
    nop                   ;[0,0]L_L_6
    LOOPSTART2
L10
[
    anda      r11,r11,r4      ;[8,1]
    anda      r9,r9,r6        ;[8,1]
    adda.lin  #<16,r9,r1      ;[0,1]
]
[
    ld.4l     (r4)+r0,d12:d13:d14:d15     ;[8,1] 1%=0 [0]
    ld.4l     (r6)+r0,d8:d9:d10:d11       ;[8,1] 0%=0 [0]
    adda.lin  #<16,r11,r8                 ;[8,1]
]
[
    sod.aaaai.4t    d8:d9,d12:d13,d8:d9        ;[8,1] 2%=0 [0]
    sod.aaaai.4t    d10:d11,d14:d15,d10:d11    ;[8,1] 2%=0 [0]
    ld.4l           (r8)+r0,d4:d5:d6:d7        ;[8,1] 2%=0 [0]
    anda            r1,r1,r2                   ;[8,1]
    ld.4l           (r1)+r0,d0:d1:d2:d3        ;[8,1] 2%=0 [0]
]
[
    anda            r10,r10,r7                 ;[8,1]
    sod.aaaai.4t    d0:d1,d4:d5,d12:d13        ;[8,1] 3%=0 [0]
    sod.aaaai.4t    d2:d3,d6:d7,d14:d15        ;[8,1] 3%=0 [0]
    adda.lin        #<16,r10,r3                ;[8,1]
    doen.3          #5                         ;[0,1] @II6
]
[
    ld.4l           (r7)+r0,d0:d1:d2:d3        ;[8,1] 3%=0 [0]
    ld.4l           (r3)+r0,d4:d5:d6:d7        ;[8,1] 1%=0 [0]
    anda            r9,r9,r5                   ;[8,1]
]
[
    mpy.l.i.4t      d2:d3,d10:d11,d2:d3        ;[8,1] 4%=0 [0]
    mpy.l.i.4t      d0:d1,d8:d9,d0:d1          ;[8,1] 4%=0 [0]
]
[
    st.4l           d0:d1:d2:d3,(r5)+r0        ;[8,1] 5%=0 [0]
    mpy.l.i.4t      d6:d7,d14:d15,d2:d3        ;[8,1] 5%=0 [0]
    mpy.l.i.4t      d4:d5,d12:d13,d0:d1        ;[8,1] 5%=0 [0]
]
    LOOPSTART3
L9
[
    ld.4l           (r6)+r0,d4:d5:d6:d7        ;[8,1] 0%=0 [1]
    ld.4l           (r4)+r0,d8:d9:d10:d11      ;[8,1] 1%=0 [1]
]
[
    sod.aaaai.4t    d4:d5,d8:d9,d12:d13        ;[8,1] 2%=0 [1]
    sod.aaaai.4t    d6:d7,d10:d11,d14:d15      ;[8,1] 2%=0 [1]
    ld.4l           (r1)+r0,d4:d5:d6:d7        ;[8,1] 2%=0 [1]
    ld.4l           (r8)+r0,d8:d9:d10:d11      ;[8,1] 2%=0 [1]
]
[
    sod.aaaai.4t    d6:d7,d10:d11,d10:d11      ;[8,1] 3%=0 [1]
    sod.aaaai.4t    d4:d5,d8:d9,d8:d9          ;[8,1] 3%=0 [1]
    ld.4l           (r7)+r0,d4:d5:d6:d7        ;[8,1] 3%=0 [1]
    st.4l           d0:d1:d2:d3,(r2)+r0        ;[8,1] 6%=1 [0]
]
[
    ld.4l           (r3)+r0,d0:d1:d2:d3        ;[8,1] 1%=0 [1]
```

```
    mpy.l.i.4t      d6:d7,d14:d15,d6:d7       ;[8,1] 4%=0 [1]
    mpy.l.i.4t      d4:d5,d12:d13,d4:d5       ;[8,1] 4%=0 [1]
]
[
    mpy.l.i.4t    d0:d1,d8:d9,d0:d1          ;[8,1] 5%=0 [1]
    mpy.l.i.4t    d2:d3,d10:d11,d2:d3        ;[8,1] 5%=0 [1]
    st.4l         d4:d5:d6:d7,(r5)+r0        ;[8,1] 5%=0 [1]
]
    LOOPEND3
[
    st.4l    d0:d1:d2:d3,(r2) ;[8,1] 6%=1 [1]
    ld.w     (r11+192),d0     ;[8,1]
]
[
    ld.w    (r9+192),d2       ;[8,1]
    ld.w    (r11+194),d5      ;[8,1]
]
[
    add.x    d2,d0,d0         ;[8,1]
    ld.w     (r9+196),d2      ;[8,1]
    ld.w     (r11+196),d3     ;[8,1]
]
[
    sxt.w.x  d0,d0            ;[8,1]
    ld.w     (r9+194),d1      ;[8,1]
    add.x    d2,d3,d2         ;[8,1]
    ld.w     (r9+198),d4      ;[8,1]
]
[
    add.x   d1,d5,d1          ;[8,1]
    ld.w    (r10+192),d5      ;[8,1]
    ld.w    (r11+198),d3      ;[8,1]
]
[
    mpy32.il.l    d0,d5,d5        ;[8,1]
    add.x         d4,d3,d4        ;[8,1]
    sxt.w.x       d1,d1           ;[8,1]
    sxt.w.x       d2,d3           ;[8,1]
    ld.w          (r10+194),d0    ;[8,1]
    ld.w          (r10+196),d2    ;[8,1]
]
[
    st.w          d5,(r9+192)     ;[8,1]
    sxt.w.x       d4,d5           ;[8,1]
    ld.w          (r10+198),d4    ;[8,1]
]
[
    mpy32.il.l    d1,d0,d1        ;[8,1]
    mpy32.il.l    d3,d2,d0        ;[8,1]
    adda.lin    #200,r10,r10    ;[0,1]
    adda.lin    #200,r11,r11    ;[0,1]
]
[
    st.w        d0,(r9+196)     ;[8,1]
    mpy32.il.l  d5,d4,d0        ;[8,1]
    st.w        d1,(r9+194)     ;[8,1]
]
[
    st.w        d0,(r9+198)      ;[8,1]
    adda.lin    #200,r9,r9       ;[0,1]
]
    LOOPEND2
    rts         ;[9,1]

    GLOBAL     F_vec_ex_end
F_vec_ex_end
FuncEnd_vec_ex
```

The listing below shows the C code used to generate the assembly file shown in the above listing.

Listing: C Code

```
#define N 100
short a[N][N], b[N][N], c[N][N];
void vec_ex() {
    int i, j;
        for (i=0; i<N; i++)
            for (j=0; j<N; j++)
            #pragma loop_unroll 2
                a[i][j] = (a[i][j] + b[i][j]) * c[i][j];
}
```

Notes

- The compiler-generated assembly files have `.sl` extension. The user-generated assembly files have `.asm` extension.
- The example in the listing above, shows the compiler-generated assembly file with compiler feedback disabled. With compiler feedback enabled, the compiler-generated assembly file contains more information and annotations. See Understanding Compiler Feedback for more information.

2.2.16 How to compile C++ source files

The StarCore compiler supports compiling `C++` source files. The basic `C++` compilation support is enabled by default; however, you need to explicitly enable the support for:

- exception handing
- double type
- the standard C++ library

To compile a `C++` source file using the command-line interface, follow these steps:

1. The compiler automatically detects a `C++` source file by interpreting the `.cpp` extension. Therefore, to compile a simple `C++` source file, specify the file name with a `.cpp` extension. The listing below shows an example `.cpp` file.
   Listing: Example .cpp file

   ```
   //file a.cpp
   #include <cstdio>

   template<typename T>

   class Complex {

   public:

     Complex( T real, T imag ): _real( real ), _imag( imag ) {}
   ```

```
  T get_real( void ) const { return _real; }

  T get_imag( void ) const { return _imag; }

  Complex operator + ( const Complex& c2 ) const { return Complex(
    _real + c2._real, _imag + c2._imag );}

private:

  T _real;

  T _imag;

};

void dump_cint( const char* str, const Complex<int>& c ) {

  std::printf( "%s=(%d,%d)\n", str, c.get_real(), c.get_imag() );

}

void main( void ) {

  Complex<int> c1( 1, 0 ), c2( 0, 1 );

  Complex<int> c3 = c1 + c2;

  dump_cint( "c1", c1 );

  dump_cint( "c2", c2 );

  dump_cint( "c3", c3 );

}
```

The listing below shows how you compile and run this `.cpp` file.

Listing: Compiling and running a .cpp file

```
$ scc -arch sc3900fp -be -mod -O3 a.cpp
$ runsim -d sc3900iss a.eld

c1=(1,0)

c2=(0,1)

c3=(1,1)
```

2. To explicitly enforce `C++` compilation for the files that do not have the `.cpp` extension, specify `-force c++` option. In case of separate compile and linking commands, the `force c++` option must always be used at the linking time in order to select the specific `C++` startup file and the `C++` linker command file. The listing below shows an example.

Listing: Forcing c++ compilation for files without .cpp extension

```
$ scc -arch sc3900fp -be -mod -O3 a.c -force c++ -o a.eln -c
$ scc -arch sc3900fp -be -mod -O3 a.eln -force c++ -o a.eld

$ runsim -d sc3900iss a.eld

c1=(1,0)

c2=(0,1)

c3=(1,1)
```

3. To enable the support for exception handing, specify the `-Cpp_exceptions on` option. To enable the support for `double` type, specify the `-slld` option. The `-slld` option must be specified both at the compile time, and the linking time. The listing below shows an example `.cpp` file.

Listing: Example .cpp file

```
//file a.cpp
#include <cstdio>

template<typename T>

class Complex {

public:

  Complex( T real, T imag ): _real( real ), _imag( imag ) {}

  T get_real( void ) const { return _real; }

  T get_imag( void ) const { return _imag; }

  Complex operator + ( const Complex& c2 ) const { return Complex(
    _real + c2._real, _imag + c2._imag );}

  Complex operator / ( const Complex& c2 ) const { throw 5; }

private:

  T _real;

  T _imag;

};

void dump_cdbl( const char* str, const Complex<double>& c ) {

std::printf( "%s=(%lf,%lf)\n", str, c.get_real(), c.get_imag() );

}

void main( void ) {

  Complex<double> c1( 1.0d, 0.0d ), c2( 0.0d, 1.0d );

  Complex<double> c3 = c1 + c2;

  try {

    Complex<double> c4 = c1 / c2;

} catch( int e ) {

    std::printf( "Caught exception!\n" );

}

  dump_cdbl( "c1", c1 );

  dump_cdbl( "c2", c2 );

  dump_cdbl( "c3", c3 );

}
```

The listing below shows how you compile and run this `.cpp` file enabling the support for exception handling and `double` type.

Listing: Enabling support for exception handling and double type

```
$ scc -arch sc3900fp -be -mod -O3 a.cpp -force c++ -o a.eln -c -
Cpp_exceptions on -slld
$ scc -arch sc3900fp -be -mod -O3 a.eln -force c++ -o a.eld    -
Cpp_exceptions on -slld

$ runsim -d sc3900iss a.eld

Caught exception!

c1=(1.000000,0.000000)

c2=(0.000000,1.000000)

c3=(1.000000,1.000000)
```

To use and compile `C++` source files in your project using the CodeWarrior IDE, select the language as **C++** on the **Build Settings** page when creating a new StarCore project.

For more information and complete list of steps to create StarCore projects using the CodeWarrior IDE, see *Targeting StarCore 3900FP DSPs*.

The new project created with **C++** language option selected is configured to:

- force C++ compilation; the `-force c++` command-line option
- enable C++ exceptions support; the `-Cpp_exceptions on` command-line option

Related Tasks

- How to use fractional data types

Related References

- Runtime libraries

2.2.17 How to disassemble the source code

Specify the `-dis` option against the project file (file with `.eld` extension) using the command-line interface. For example:

```
disasmsc100 -arch sc3900fp my_eld_file.eld
```

The compiler prints the disassembly of the entire `.eld` file. Output is in the order of sections as they are placed in the `.eld` file and not in the address low to address high order.

The listing below shows an example of a disassembled code. The disassembled code is truncated to an appropriate length.

Listing: Example of disassembled code

```
;Global EQUs
__exception_table_start__     equ     $0

TIMER     equ     $0

CPlusPlus     equ     $0

ENABLE_EXCEPTION     equ     $0

___exec_mrom_decompression     equ     $0

ROM_INIT     equ     $0

__exception_table_end__     equ     $0

_VBAddr     equ     $0

_SR_Setting     equ     $c

ARGV_LENGTH     equ     $100

_CodeStart     equ     $1c00

_cpp_staticinit_end     equ     $af10

_cpp_staticinit_start     equ     $af10

_SR2_Setting         equ     $1f0041

__BottomOfHeap     equ     $40000000

_StackStart     equ     $40000000

_TopOfStack     equ     $7fffeff0

__TopOfHeap     equ     $7fffeff0

_ROMStart     equ     $80000000

;Local EQUs

        section memory_intr_vec

        sectype progbits

        secflags alloc

        secflags execinstr

        align 64

        global     TRAP0_exception

        global     IntVec

        global     TRAP1_exception

;00000000:

F__MemAllocArea_2_00000000

.rela.line.debug_info.DW_AT_low_pc
```

```
TRAP0_exception

IntVec

            jmp $1c68        ;($00001c68=___crt0_start)

;00000006:

F__MemAllocArea_2_00000006

            nop

;00000008:

F__MemAllocArea_2_00000008

            nop

;0000000a:

F__MemAllocArea_2_0000000a

            nop

;0000000c:

F__MemAllocArea_2_0000000c

            nop

;0000000e:

F__MemAllocArea_2_0000000e

            nop

;00000010:

F__MemAllocArea_2_00000010

            nop

;00000012:

F__MemAllocArea_2_00000012

            nop

;00000014:

F__MemAllocArea_2_00000014

            nop

;00000016:

F__MemAllocArea_2_00000016

            nop

;00000018:

F__MemAllocArea_2_00000018

            nop

;0000001a:

F__MemAllocArea_2_0000001a
```

```
            nop
;0000001c:
F__MemAllocArea_2_0000001c
            nop
;0000001e:
F__MemAllocArea_2_0000001e
            nop
;00000020:
F__MemAllocArea_2_00000020
TRAP1_exception
            jmp $1c64         ;($00001c64=___EmptyIntHandler)
;00000026:
F__MemAllocArea_2_00000026
            nop
;00000028:
F__MemAllocArea_2_00000028
            nop
;0000002a:
F__MemAllocArea_2_0000002a
            nop
;0000002c:
F__MemAllocArea_2_0000002c
            nop
;0000002e:
F__MemAllocArea_2_0000002e
            nop
;00000030:
F__MemAllocArea_2_00000030
            nop
;00000032:
F__MemAllocArea_2_00000032
            nop
;00000034:
F__MemAllocArea_2_00000034
            nop
```

```
;00000036:
F__MemAllocArea_2_00000036
            nop
;00000038:
F__MemAllocArea_2_00000038
            nop
;0000003a:
F__MemAllocArea_2_0000003a
            nop
;0000003c:
F__MemAllocArea_2_0000003c
            nop
;0000003e:
F__MemAllocArea_2_0000003e
            nop
;00000040:
.rela.line.debug_info.DW_AT_high_pc
        endsec
```

Figure 2-1. Example of disassembled code

Notes

- The source code that is not assembled appears in the comment form
- Disassembled output can be modified and re-assembled if required

Related Tasks

- How to keep compiler-generated assembly files

2.2.18 How to use emulation library

Each emulation in the emulation library is mapped to its counterpart intrinsic available in the supported platform and tool chain.

Once the emulation library is correctly integrated, the code written using StarCore intrinsics, and compiled and run in one of the supported platforms, yields the same result as it does in the sole StarCore environment.

Related Concept

- Understanding emulation library

Related Reference

- Intrinsics supported in emulation library

2.2.18.1 Using emulation library on Windows® platform

Currently, only the Visual Studio and the CL compiler are supported on the Windows platform.

To use emulation library with Visual Studio:

NOTE

The steps described below to use emulation library are indicative only. The actual steps may vary depending upon the version of Visual Studio in use. For additional details, refer the Visual Studio help.

1. Include header file, `prototype.h` for SC3850 and `prototype_sc3900.h` for SC3900FP, in your project, using either of the following ways:
   - Drag-and-drop the header file in the **Header Files** directory in the **Solution Explorer** view.
   - Using **Options** dialog box:
     1. From the menu bar, select **Tools > Options**.

        The **Options** dialog box appears.

     2. Expand the **Projects and Solutions** node and select the **VC++ Directories** property page.
     3. Select **Include files** from the **Show directories for** drop-down list.
     4. Click the **Folder** icon above the list of directories. A blank line is added in the directory list.
     5. Click the **Ellipsis** button in the blank line. Browse to and select the directory containing the header file.
2. Using project property pages:
   a. Right-click your project in the **Solution Explorer** view, and select **Properties**.

      The project property pages appear.

   b. Select **Configuration Properties -> C/C++ -> General**.

c. In the **Additional Include Directories** property, add the directory containing the header file.

3. Include library file, `prototype.lib`, in your project:
   a. From the menu bar, select **Tools > Options**.

      The **Options** dialog box appears.

   b. Expand the **Projects and Solutions** node and select the **VC++ Directories** property page.
   c. Select **Library files** from the **Show directories for** drop-down list.
   d. Click the **Folder** icon above the list of directories. A blank line is added in the directory list.
   e. Click the **Ellipsis** button in the blank line, and browse to and select the directory containing the library file.
   f. Right-click your project in the **Solution Explorer** view, and select **Properties**.

      The project property pages appear.

   g. Select **Configuration Properties > Linker > Input**.
   h. In the **Additional Dependencies** property, enter `prototype.lib`.
4. Include the DLL file, `prototype.dll`, in your project. The DLL file lets you debug and run the project code.
   a. Right-click your project in the **Solution Explorer** view, and select **Properties**.

      The project property pages appear.

   b. Select **Configuration Properties > Debugging**.
   c. In the **Environment** property, type `PATH=`*<absolute_path_to_dll>*

To use emulation library with CL compiler at command-line:

1. Include reference to the `prototype.lib` file while compiling the project.

   ```
   cl project.c /link <path_to_the_library>\prototype.lib
   ```

2. Ensure that `prototype.dll` file is specified in PATH variable or local directory while executing the project.
3. Include reference to the `prototype.h` file, or make the file available in local directory.

2.2.18.2 Using emulation library on Linux platform

Currently, only gcc is supported on the Linux platform.

To use emulation library with gcc:

1. Copy `prototype.lib` locally.

```
cp <path_to_compiler>/library/libprototype.so.1.0
```

2. The Linux version of the library is a dynamically linked shared object library. So, you need to create two soft links.
   a. Create the link to `libprototype.so` to allow the naming convention for the compiler flag `-lprototype` to work in the compilation command line.

```
ln -s libprototype.so.1.0 libprototype.so #or /usr/lib/ libprototype.so.
1.0 /usr/lib/libprototype.so
```

   b. Create the link to `libprototype.so.1` to allow the run time binding to work.

```
ln -s libprototype.so.1.0 libprototype.so.1 #or /usr/lib/ libprototype.so.
1.0 /usr/lib/libprototype.so.1
```

3. Compile the project, `Project1`.

```
gcc -O3 project1.c -I<path to include file> -lprototype -o project
```

NOTE

In case the prototype header is copied locally, `-I<path to include file>` is no longer necessary in the command line

4. Execute the project.

```
./project1
```

2.2.19 How to use MEX-library in MATLAB® environment

To use the MEX-library in the MATLAB® environment, follow these steps:

1. Navigate to following sub-directory in the CodeWarrior installation directory:

   `SC/StarCore_Support/compiler/library`
2. Locate the `prototype.dll` and the `sc3900_mex_library.zip` file.
3. The path for libraries in the MATLAB® environment must point to the `prototype.dll` and the contents of `sc3900_mex_library.zip` file. You can either:
   - unzip the `sc3900_mex_library.zip` file in-place, and then add the path (identified in Step 1) in the MATLAB® environment, or
   - copy the `prototype.dll` file and unzip the `sc3900_mex_library.zip` file to a new directory, and then add this new directory's path in the MATLAB® environment.

The MEX-library can be called like any other MATLAB® function. For instance, to add two 32-bit values, use the following function:

```
>> sc3900__l_add_x( 1, 2)
```

Related Concept

- Understanding MEX-library

Related Reference

- Intrinsics supported in emulation and MEX library

2.2.20 How to enable code reordering

To enable the compiler for reordering the code for better performance, use the `likely` and `unlikely` macros. The purpose is to mark any unlikely executed code so that it can be moved away in the generated code.

You can apply the macros on `if` statements as follows:

Marking an `if` condition as `unlikely`

```
if (unlikely(a > 1))
```

Marking an `else` branch as `unlikely`

```
if (likely(a <= 1))
{
}
else // this will be considered unlikely
{
}
```

Related Concepts

- Understanding code reordering

2.2.21 How to disable automatic vectorization

To fully disable the automatic vectorization at loop level, add the pragma `#pragma novector` in the considered loop code. To fully disable the vectorization at function level, add the attribute `__attribute__((novector))` to the function.

Related References

- #pragma novector and __attribute__((novector))

2.3 Pragmas and attributes tasks

In this section:

- Function pragmas and attributes tasks
- Statement pragmas tasks
- Other pragmas and attributes tasks

2.3.1 Function pragmas and attributes tasks

In this section:

- How to align function
- How to align structure fields
- How to specify section attribute
- How to allow non-standard returns
- How to define function as interrupt handler
- How to allow compiler to perform load speculation above loop breaks

2.3.1.1 How to align function

The compiler follows one function-alignment rule if you optimize for code speed, but a different rule if you optimize for code size. To override either default rule, use `__attribute__((aligned(<n>)))` at the beginning of a function. This pragma also overrides function alignment given through an application file.

The listing below shows an example of this attribute, with alignment value 1024.

Listing: __attribute__((aligned(<n>)))

```
void alpha()
__attribute__((aligned(1024)))

{

    printf("Hello!\n"];

}
```

Compiling these files together (`scc -ma align.appli alpha.c ...`) yields the same alignment as the `__attribute__((aligned(<n>)))` example in the above listing.

Related Concepts

- Pragmas and attributes concepts

Related References

- Function pragmas and attributes
- Pragmas to control function inlining

2.3.1.2 How to align structure fields

You can align the structure fields using __attribute__((aligned(x))) option. The listing below shows an example.

Listing: __attribute__((aligned(x))) example

```
typedef struct
{

    short a;

    short b;

    short c;

    __attribute__((aligned(8))) short test[16];

    short * ptr;

} OK1_T;
```

The listing below shows how you use the `__attribute__((packed))` option to specify packing for the members of a structure.

Listing: __attribute__((packed)) example

```
typedef struct __attribute__((packed)) z {
    unsigned short m0;

    unsigned int   m1;

} z_struct;
```

The attribute option in the above listing applies to the members and it also specifies 1 byte packing. Therefore, if `m0` is changed to a `char` data type, you get a struct size of 5 with `m1` at offset 1. If you want a 2 byte packed alignment of the struct without changing the alignment of the members you would use the

`__attribute__((packed, aligned(n)))` option.

The listing below shows an example.

Listing: __attribute__((packed, aligned(n))) example

```
typedef struct  x {
    unsigned short m0;

    unsigned int   m1;

} __attribute__((packed, aligned(2))) x_struct;
```

Related Concepts

- Pragmas and attributes concepts

Related References

- Function pragmas and attributes
- Pragmas to control function inlining

2.3.1.3 How to specify section attribute

The new section attribute lets you place a data object or a function in a specific section. The syntax is: __attribute__((section("section_name"))), where section_name specifies the target section's name.

The target section can be:

- a basic section, such as data, program, bss, rom, or init section
- a custom section, defined in the application file
- a previously undefined section

If the target section is a previously undefined section, the compiler creates a new section, and the section type depends on the current context.

Related Concepts

- Pragmas and attributes concepts

Related References

- Function pragmas and attributes
- Pragmas to control function inlining

2.3.1.4 How to allow non-standard returns

Use `__attribute__((noreturn))` to permit non-standard returns from functions to their calling routines. Ordinarily, the compiler cannot implement optimizations that involve removing function code that includes the normal return instructions. But either of these pragmas make such optimizations possible, by letting the compiler abort the function safely.

- Use `__attribute__((noreturn))` in a header file, or at the head of a source file. This attribute specifies its function, so does not need to be in the same module as the function definition.
- Use `__attribute__((noreturn))` inside the definition of its function.

Related Concepts

- Pragmas and attributes concepts

Related References

- Function pragmas and attributes
- Pragmas to control function inlining

2.3.1.5 How to define function as interrupt handler

A function that operates as an interrupt handler differs from other functions:

- It must save and restore all registers, without assuming any conventions, as the interrupt handler may be called at any time.
- It must not receive passed parameters; it must not return a value.

To define a function as an interrupt handler, use `__attribute__ ((interrupt0/interrupt1/interrupt2))`. Another option is to use `#pragma interrupt`, which is equivalent to `__attribute__(interrupt0)`), as the listing below shows.

Listing: #pragma interrupt

```
void IntHandler();
#pragma interrupt IntHandler

extern long Counter;
```

```
void IntHandler()
{
    Counter++;
}
```

Note that beginning build 24.1.1 of the StarCore Compilers, attributes are preferred for defining a function as an interrupt handler.

The SC3900FP core programming model consists of three link registers, and the compiler provides three corresponding attributes, one for each link register, that can be used to define a function as an interrupt handler.

- `interrupt0`, equivalent with `#pragma interrupt` and with the attribute `interrupt`. It can be used for all exceptions and interrupts except for MMU and debug exceptions. Related to the LR0 link register.

  Example:

  ```
  void foo0(int *a) __attribute__((interrupt0))
  ```
- `interrupt1`, used for MMU exceptions only, related to the LR1 link register.

  ```
  void foo1(int *a) __attribute__((interrupt1))
  ```
- `interrupt2`, used for Debug exceptions only, related to the LR2 link register.

  ```
  void foo2(int *a) __attribute__
  ((interrupt2))
  ((interrupt2))
  ```

Related Concepts

- Pragmas and attributes concepts

Related References

- Function pragmas and attributes
- Pragmas to control function inlining
- Pragmas to control function inlining

2.3.1.6 How to allow compiler to perform load speculation above loop breaks

To let the compiler know that it is safe to speculate loads above breaks, use `#pragma load_speculation` inside the required loop.

The listing below shows an example.

Listing: #pragma load_speculation - Example

```
int k, s = 0;
for( k = 0; k < SIZE; k ++ )

{

#pragma load_speculation

s += B[0][k] * C[k][0];

if (s < 0 ) {

break;

}

}
```

Generated code:

- Without `#pragma load_speculation`, the compiler cannot perform software pipelining:

```
LOOPSTART0

L8

[

ld.l            (r15)+,d13

ld.l            (r18)+r17,d27

]

mac32.il.l      d13,d27,d12

cmp.gt.l        #<-1,d12,p0:p1

IF.p1       break.0       L9

nop

LOOPEND0
```

- With `#pragma load_speculation`, the compiler can move memory reads (LOAD) above the break and software pipelining can be applied:

```
LOOPSTART0

L8
```

```
ld.l            (r15)+,d13          ; 0%=0 [1]

ld.l            (r18)+r17,d27       ; 0%=0 [1]

IF.p1       break.0       PL004               ; 6%=1 [0]

]

mac32.il.l      d13,d27,d12         ; 1%=0 [1]

cmp.gt.l        #<-1,d12,p0:p1      ; 2%=0 [1]

LOOPEND0
```

2.3.2 Statement pragmas tasks

In this section:

- How to specify profile value
- How to define loop count
- How to map switch statements

2.3.2.1 How to specify profile value

The default settings of your compiler's profiler determine the number of times to execute a given statement. To override this default guidance, specifying the exact number of times to execute a statement, use `#pragma profile`.

NOTE

The value argument of `#pragma profile` statement must be a 32-bit signed integer. The maximum value is `0x7FFFFFFF`.

In the listing below, `#pragma profile` tells the compiler to execute the loop only 10 times. But note that this loop has dynamic bounds - if `#pragma profile` were not present, the compiler would execute the loop 25 times (the default value for such loops). The 15 additional loop executions would not make the program results incorrect, but would impair optimization.

Listing: #pragma profile with constant value

```
#include <prototype_sc3900.h>
int energy (short block[], int N)

{

int i;

long int L_tmp = 0, tmp;

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

#pragma profile 10

tmp=__l_put_msb(block[i]);

L_tmp = __l_mac_r_x_hh (L_tmp, tmp, tmp);

}

return L_tmp;

}
```

With `if-then-else` constructs, you can use `#pragma profile` to tell the compiler which branch executes more frequently. The pragma also can convey the frequency ratio - the number of times one branch executes in relation to the other branch.

In the listing below, for example, the two `#pragma profile` statements have values 5 and 50. These values tell the compiler that the `else` branch section executes 10 times more frequently than the first (implied `then`) branch section. For this use, the frequency ratio is more significant than the exact `#pragma profile` values. In this example, the values 1 and 10 would convey the same information.

Listing: #pragma profile with frequency ratio

```
#include <prototype_sc3900.h>
int energy (short block[], int N)

{

int i;

long int L_tmp = 0,tmp;

if ( N>50)

#pragma profile 5

for (i = 0; i < 50; i++){

tmp=__l_put_msb(block[i]);

L_tmp = __l_mac_r_x_hh (L_tmp, tmp, tmp);

}

else

#pragma profile 50

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

tmp=__l_put_msb(block[i]);
```

```
L_tmp = __l_mac_r_x_hh (L_tmp, tmp, tmp);
}
return L_tmp;
}
```

Related Concepts

- Pragmas and attributes concepts

Related References

- Statement pragmas and attributes

2.3.2.2 How to define loop count

If static information is available, the compiler uses it to determine the number of iterations for a loop. If static information is not available, the compiler still can determine the number of loop iterations, provided that it knows the upper and lower iteration limits. To provide these limits to the compiler, use `#pragma loop_count`.

For example, the compiler can use the lower iteration limit to remove loop bypass tests. It can use the upper iteration limit to assess the range of induction variables after the loop, even if the exact loop-count value cannot be known at compile time.

Specifying a modulo divider - 2 or 4 - enables the optimizer to unroll loops in the most efficient way. This modulo value corresponds to the number of execution units. You also can tell the compiler whether to use any remainder to execute the loop additional times. The compiler can use the modulo/remainder pair to unroll loops with dynamic loop count.

The syntax of this pragma is:

```
#pragma loop_count (min_iter, max_iter,[{modulo}, [remainder]])
```

Note that:

- The modulo parameter is optional; the only valid values are 2 or 4.
- To tell the compiler to use a remainder for the loop count, specify a value for `remainder`. A `remainder` argument is valid only if you specify a divider value.
- You must place `#pragma loop_count` inside its loop, but outside any subordinate, nested loops.

In the code listing below, the loop iterates at least 100 times, at most 512 times. The iteration count always will be divisible by 8. The pragma does not specify a remainder, so the compiler disregards any remainder from the division.

Listing: #pragma loop count

```
#include <prototype_sc3900.h> #define N 200

short a[N], b[N], c[N];

void vec_ex(int n) {

    int i;

    for (i=0; i<n; i++)

        #pragma loop_count (100,512,8,0)

    a[i] = a[i]+b[i]*c[i];

}
```

Related Concepts

- Pragmas and attributes concepts

Related References

- Statement pragmas and attributes

2.3.2.3 How to map switch statements

Two representations are available for mapping switch statements onto StarCore assembly code. They are based on:

- A computed `goto` - this requires a table of switch cases, stored in a RAM `.data` section.
- A succession of conditional branches, with static conditions.

The optimization type (speed or size) guides the compiler in selecting the appropriate representation, as does the footprint of the switch cases. If there are only a few successive switch cases, computed `goto` mapping is superior. But if the switch-case table is large, with many holes, conditional-branch mapping yields smaller code size.

To force either type of mapping, use the appropriate pragma just before the switch statement:

- `#pragma switchtable` - forces computed `goto` mapping, provided that code generated for lines between the switch-statement braces ({ }) fits into 32-bit integers.

- `#pragma switchtablebyte` - forces computed `goto` mapping, provided that code generated for lines between the switch-statement braces ({ }) fits into 255 bytes.
- `#pragma switchtableword` - forces computed `goto` mapping, provided that code generated for lines between the switch-statement braces ({ }) fits into 65535 bytes.
- For conditional-branch mapping, use `#pragma noswitchtable`.

NOTE

The default location for switchtables is the `.data` section in RAM. To move them to ROM, instead, set the `Switch_To_Rom` option, in the configuration application file.

For the example code of the listing displayed below, the compiler executes one of three pragmas, according to the values of NOSWITCH and USEWORD:

- If the compiler executes `#pragma noswitchtable`, it produces 2746 bytes of code and 434 bytes of data (no switch table).
- If the compiler executes `#pragma switchtable`, it produces 1964 bytes of code and 834 bytes of data (an integer-type switch table).
- If the compiler executes `#pragma switchtableword`, it produces 1964 bytes of code and 634 bytes of data (a word-type switch table).

Listing: #pragmas noswitchtable, switchtable, switchtableword

```
#include <stdio.h>
int Tab[100];

void alpha(int i)

__attribute__ ((noinline))

{

#ifdef NOSWITCH

#pragma noswitchtable

#else

#pragma switchtable

#ifdef USEWORD

#pragma switchtableword

#endif

#endif

        switch (i) {

                case 0: Tab[0] = 0; break;

                case 1: Tab[1] = 1; break;

                case 2: Tab[2] = 2; break;

                case 3: Tab[3] = 3; break;
```

```
case 4: Tab[4] = 4; break;
case 5: Tab[5] = 5; break;
case 6: Tab[6] = 6; break;
case 7: Tab[7] = 7; break;
case 8: Tab[8] = 8; break;
case 9: Tab[9] = 9; break;
case 10: Tab[10] = 10; break;
case 11: Tab[11] = 11; break;
case 12: Tab[12] = 12; break;
case 13: Tab[13] = 13; break;
case 14: Tab[14] = 14; break;
case 15: Tab[15] = 15; break;
case 16: Tab[16] = 16; break;
case 17: Tab[17] = 17; break;
case 18: Tab[18] = 18; break;
case 19: Tab[19] = 19; break;
case 20: Tab[20] = 20; break;
case 21: Tab[21] = 21; break;
case 22: Tab[22] = 22; break;
case 23: Tab[23] = 23; break;
case 24: Tab[24] = 24; break;
case 25: Tab[25] = 25; break;
case 26: Tab[26] = 26; break;
case 27: Tab[27] = 27; break;
case 28: Tab[28] = 28; break;
case 29: Tab[29] = 29; break;
case 30: Tab[30] = 30; break;
case 31: Tab[31] = 31; break;
case 32: Tab[32] = 32; break;
case 33: Tab[33] = 33; break;
case 34: Tab[34] = 34; break;
case 35: Tab[35] = 35; break;
case 36: Tab[36] = 36; break;
case 37: Tab[37] = 37; break;
case 38: Tab[38] = 38; break;
```

```
case 39: Tab[39] = 39; break;
case 40: Tab[40] = 40; break;
case 41: Tab[41] = 41; break;
case 42: Tab[42] = 42; break;
case 43: Tab[43] = 43; break;
case 44: Tab[44] = 44; break;
case 45: Tab[45] = 45; break;
case 46: Tab[46] = 46; break;
case 47: Tab[47] = 47; break;
case 48: Tab[48] = 48; break;
case 49: Tab[49] = 49; break;
case 50: Tab[50] = 50; break;
case 51: Tab[51] = 51; break;
case 52: Tab[52] = 52; break;
case 53: Tab[53] = 53; break;
case 54: Tab[54] = 54; break;
case 55: Tab[55] = 55; break;
case 56: Tab[56] = 56; break;
case 57: Tab[57] = 57; break;
case 58: Tab[58] = 58; break;
case 59: Tab[59] = 59; break;
case 60: Tab[60] = 60; break;
case 61: Tab[61] = 61; break;
case 62: Tab[62] = 62; break;
case 63: Tab[63] = 63; break;
case 64: Tab[64] = 64; break;
case 65: Tab[65] = 65; break;
case 66: Tab[66] = 66; break;
case 67: Tab[67] = 67; break;
case 68: Tab[68] = 68; break;
case 69: Tab[69] = 69; break;
case 70: Tab[70] = 70; break;
case 71: Tab[71] = 71; break;
case 72: Tab[72] = 72; break;
case 73: Tab[73] = 73; break;
```

```
                case 74: Tab[74] = 74; break;
                case 75: Tab[75] = 75; break;
                case 76: Tab[76] = 76; break;
                case 77: Tab[77] = 77; break;
                case 78: Tab[78] = 78; break;
                case 79: Tab[79] = 79; break;
                case 80: Tab[80] = 80; break;
                case 81: Tab[81] = 81; break;
                case 82: Tab[82] = 82; break;
                case 83: Tab[83] = 83; break;
                case 84: Tab[84] = 84; break;
                case 85: Tab[85] = 85; break;
                case 86: Tab[86] = 86; break;
                case 87: Tab[87] = 87; break;
                case 88: Tab[88] = 88; break;
                case 89: Tab[89] = 89; break;
                case 90: Tab[90] = 90; break;
                case 91: Tab[91] = 91; break;
                case 92: Tab[92] = 92; break;
                case 93: Tab[93] = 93; break;
                case 94: Tab[94] = 94; break;
                case 95: Tab[95] = 95; break;
                case 96: Tab[96] = 96; break;
                case 97: Tab[97] = 97; break;
                case 98: Tab[98] = 98; break;
                case 99: Tab[99] = 99; break;
                default: Tab[1] = 1; break
        }
}
void main()
{
        int i;
        for (i = 0; i<100; i++) alpha(i);
        for i = 0; i<100; i++) if (Tab[i] != i)
                {printf("Failed at %d\n", i); exit(0);}
```

```
        i = clock();

        printf("Passes [%d cycles]\n", i);
    }
```

Related Concepts

- Pragmas and attributes concepts

Related References

- Statement pragmas and attributes

2.3.3 Other pragmas and attributes tasks

In this section:

- How to align variables
- How to specify optimization level
- How to rename segments of ELF files
- How to unroll and jam loop nest
- How to control GNU C/C++ extensions

2.3.3.1 How to align variables

Size usually determines object alignment; the default array alignment is by base type. But certain external functions require array alignment to a specified value before you can pass the array to the function.

Use `__attribute__((aligned(<n>)))` to force such alignment, specifying the defined array object and the value 4 or 8. The value 4 specifies 4-byte (32-bit double word) alignment; the value 8 specifies 8-byte (64-bit quad word) alignment.

The syntax is:

```
int __attribute__((aligned(8))) T1[100]
```

or

```
int *__attribute__((aligned(8)))
```

For example, `int __attribute__((aligned(8))) T1[100]` specifies 8-byte alignment for array T1.

`int *__attribute__((aligned(8))) a` tells the compiler that `*a` points to the alignment information. It also tells the compiler to use 8-byte alignment, if possible.

In the first part of the listing below, array `a` is forced to 8-byte alignment before being passed to the external function `Energy`. The second part of the example informs the compiler that both input vectors are aligned to 32 bits. Aligning variables is required for accessing a semaphore.

Listing: #pragma align

```
#include <prototype_sc3900.h>
short __attribute__((aligned(8))) a[10];

extern int Energy( short a[] );

int alpha()

{

return Energy(a);

}

long int Cor(short __attribute__((aligned(4)))  vec1[], short
__attribute__((aligned(4))) vec2[], int N)

{

long int L_tmp = 0;

long int L_tmp2 = 0;

int i;

for (i = 0; i < N; i += 2){

L_tmp = L_tmp + vec1[i]*vec2[i];

L_tmp2 = L_tmp2 + vec1[i+1]* vec2[i+1];

}

return L_tmp + L_tmp2;

}
```

Related Concepts

- Pragmas and attributes concepts

Related References

- Other pragmas and attributes

2.3.3.2 How to specify optimization level

The opt_level pragma can apply to a single function or to the whole module. To apply opt_level to a function, place the pragma in the function body. To apply opt_level to a module, place the pragma at the module level.

An opt_level pragma in a function supersedes an opt_level pragma at the module level. An opt_level pragma at the module level supersedes the optimization level the shell passes.

The listing below shows the possible opt_level pragma statements.

Listing: Possible opt_level pragma statements

```
# This statement equals scc -O0.
#pragma opt_level = "O0"

# This statement equals scc -O1.

#pragma opt_level = "O1"

# This statement equals scc -O2.

#pragma opt_level = "O2"

# This statement equals scc -O3.

#pragma opt_level = "O3"

# This statement equals scc -Os.

#pragma opt_level = "Os"

# This statement equals scc -Os -O3.

#pragma opt_level = "O3s"
```

You cannot use -O3 as a command-level option with the O0, O1, O2, and Os options; you can use -O3 only with O3s.

At command level, the O0, O1, O2, and Os options are compatible with O0O1O2Os as pragmas.

The listing below shows a code example that uses the opt_level pragma. If the command-line is scc -Os opt.c, the compiler compiles func1 in O0 as the module-level option is O0. The compiler compiles the func2 function in O2 (which overrides O0 specified in the module and Os specified in the command line).

Listing: Opt.c: opt_level pragma code example

```
typedef struct {
    int a;

    int b;

} S;

#pragma opt_level = "O0"

void func1()

{

    typedef struct {
```

```
        short a;
        short b;
    } S;
    S v;
    v.a = 0;
    v.b = 1;
}
void func2()
{
#pragma opt_level = "O2"
    S v;
    v.a = 2;
    v.b = 3;
}
```

Related Concepts

- Pragmas and attributes concepts

Related References

- Other pragmas and attributes

2.3.3.3 How to rename segments of ELF files

Five pragmas let you rename segments or ELF files:

- `#pragma pgm_seg_name "name"{, "overlay"}` - to rename a text segment.
- `#pragma data_seg_name "name"` - to rename a data segment.
- `#pragma rom_seg_name "name"{, "overlay"}` - to rename the ROM segment.
- `#pragma bss_seg_name "name"` - to rename the bss segment.
- `#pragma init_seg_name "name"` - to rename the init segment.

For any of these pragmas, you must follow these rules:

- The new segment name you define cannot include any spaces.
- You can place the pragma anywhere in the module (file); it affects the entire file.
- You must define the name used to override the default segment name in the linker command file.

NOTE

To specify text-segment overlay, add an overlay value to the segment-name or rom-segment-name pragma .

Related Concepts

- Pragmas and attributes concepts

Related References

- Other pragmas and attributes

2.3.3.4 How to unroll and jam loop nest

The unroll-and-jam transformation consists of unrolling an enclosing loop a specified number of times, then merging multiple instances of the enclosed loop. The unroll-and-jam factor specifies the number of times to unroll the enclosing loop.

The goal of this transformation is increasing the parallelism available for pipelining. However, this transformation changes the order of execution for the loop iterations. This means that unrolling-and-jamming is not always valid: it could violate data dependencies.

The following listing shows the syntax of this pragma.

Listing: Unroll and jam pragma syntax

```
...
for() {
  for() {
    #pragma loop_unroll_and_jam factor
    ...
  }
}
...
```

Related Concepts

- Pragmas and attributes concepts

Related References

- Other pragmas and attributes

2.3.3.5 How to control GNU C/C++ extensions

The `gcc_extensions` pragma controls the GNU C/C++ extensions.

Syntax

```
#pragma gcc_extensions on | off | reset
```

Remarks

If you enable this pragma, the compiler accepts GNU C/C++ extensions in the source code. This includes the following non-ANSI C/C++ extensions:

- Initialization of automatic `struct` or `array` variables with non- `const` values.
- Illegal pointer conversions
- `sizeof( void ) == 1`
- `sizeof( function-type ) == 1`
- Limited support for GCC statements and declarations within expressions.
- Macro redefinitions without a previous `#undef`.
- The GCC keyword `typeof`
- Function pointer arithmetic supported
- `void*` arithmetic supported
- Void expressions in return statements of void
- `__builtin_constant_p (`*expr*`)` supported
- Forward declarations of arrays of incomplete type
- Forward declarations of empty static arrays
- Pre-C99 designated initializer syntax (deprecated)
- shortened conditional expression (*c* `?:` *y*)
- `long __builtin_expect (long exp, long c)` now accepted

Related Concepts

- Pragmas and attributes concepts

Related References

- Other pragmas and attributes

Chapter 3
Concepts

This chapter consists of compiler concepts that you might need to comprehend to accomplish compiler tasks.

In this chapter:

3.1 Compiler configuration concepts

In this section:

3.1.1 Understanding compiler environment

The StarCore compiler supports various file types. Table 3-1 lists the filename extensions and their corresponding file types. The table also lists which component processes each file type.

Table 3-1. Compiler file types and extensions

File Extension	File Type	Component
`.c/c++`	C/C++ source file	Preprocessor
`.h`	C header file	
`w.i`	Preprocessed C source	Front End
`.lib`	IR library	Optimizer
`.sl`	IR linear assembly file	Optimizer
`.asm, .sl`	Assembly file	Assembler
`.eln`	Relocatable ELF object file	Linker
`.cmd, .mem, .l3k`	Linker command file	Linker

NOTE

It is possible to force the shell to process a file as if it were a different file type.

The final result of the compilation process is an executable object file, which has the `.eld` extension. Figure 3-1 shows filename extensions at each stages of the compilation process.

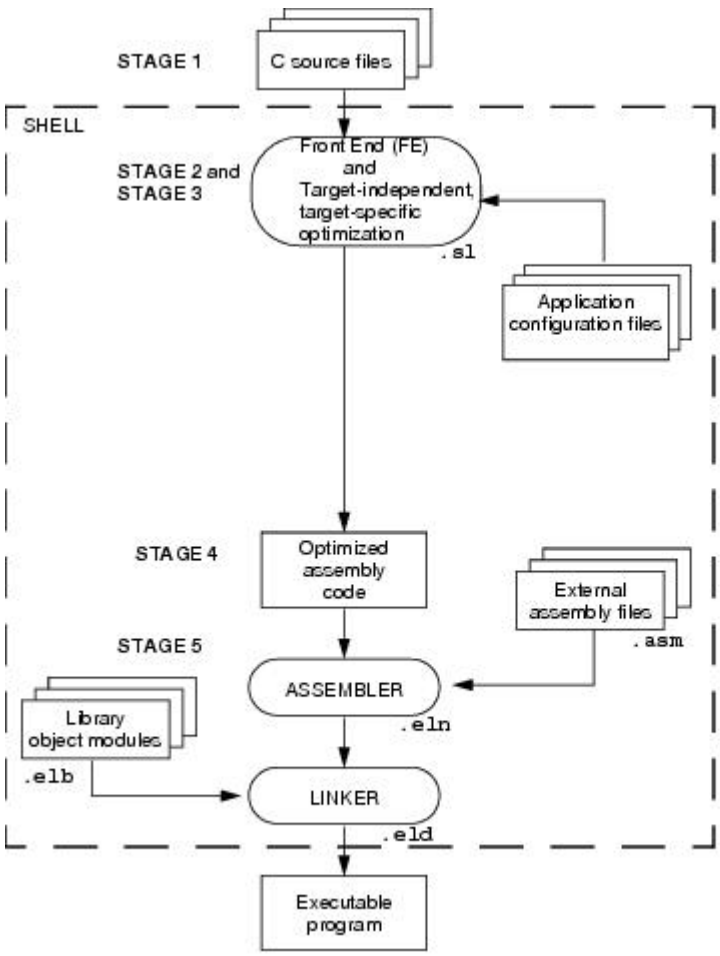

Figure 3-1. Compilation stages

Table 3-2 describes the stages in detail.

Table 3-2. Description of compilation stages

Stage	Description
Stage 1	You invoke the shell, specifying the C/C++ source files and assembly files to be processed, and specifying the appropriate options
Stage 2 and Stage 3	The Front End (FE) identifies C/C++ source files by their file extension, pre-processes the source files, then passes the files to the optimizer. In the high-level optimization phase, the optimizer translates each IR file to an assembly ASCII file, and then performs target-independent optimizations. The optimizer can extract library files created in the IR form, including these library files at this stage of processing. Additionally, optimization includes any relevant information that the application and the machine-configuration files contain. The output of this stage is linear assembly code.

Table continues on the next page...

Table 3-2. Description of compilation stages (continued)

Stage	Description
Stage 4	In the low-level optimizer phase, the optimizer carries out target-specific optimizations. It transforms the linear assembly code to the parallel assembly code.
Stage 5	The optimizer sends the optimized assembly files to the assembler. The assembler assembles these files together with any specified external assembly files, and outputs these files to the linker. The linker combines the assembly object files together with any specified external assembly files, extracts any required object modules from the library, and produces the executable application.

If none of the compiler stages recognize a filename extension, the compiler treats the file as an input file for the linker.

The shell provides a one-step command-line interface that you use to specify the files for each stage of the compilation process.

Note that beginning build 23.10.2 of the StarCore compiler, the default linker is sc3000-ld. In addition:

- the valid extension for sc3000-ld is only `.l3k`

Related Tasks

- How to set environment variables
- How to create user-defined compiler startup code file

Related Concepts

- Understanding compiler startup code

3.1.2 Understanding compiler startup code

The compiler startup code consists of following two phases:

- Bareboard startup code phase; an optional phase. The programs that execute without the support of any runtime executive or operating system run through this phase. In this phase, the compiler:
  - resets the interrupt vector table to default arrangement; sets the first entry of the table to the system entry point, `__crt0_start`, and rest of the entries to the `abort` function.
  - initializes hardware registers

  - activates the timer, if any
  - terminates the bareboard startup code phase and jumps to C/C++ environment startup code entry point, `___start`
- C/C++ environment startup code phase; a mandatory phase of all programs. This phase includes the initialization code and the finalization code. The initialization code is executed after the program initiates and before the `main` function is called. By default, the initialization code:
  - sets up and initializes the memory map
  - copies the initialized variables from ROM to RAM, if the `-mrom` option is specified. This option is required for the applications that do not use a `loader`.
  - sets the `argv` and `argc` parameters
  - enables interrupts, if disabled
  - calls the `main` function

  The finalization code is executed after the `main` function terminates. By default, the finalization code uses the `exit` function to terminate the I/O services that the application did not terminate. The `exit` function also executes the `stop` instruction to stop the processor.

  The C/C++ environment startup code also provides low-level, buffered I/O services to interact with debugging tools and run-time systems.

Related Tasks

- How to set environment variables
- How to create user-defined compiler startup code file

Related Concepts

- Understanding compiler environment

3.1.3 Understanding memory models

The StarCore compiler supports following memory models:

- Small memory model

  Enabled by default; appropriate when all static data fits into the lower 64KB of the address space. All addresses must be 16-bit addresses.

- Big memory model

Appropriate when code, static data, and runtime library require 64KB to 1MB of memory. The big memory model does not restrict the memory space that is allocated to the addresses. In addition, when the big memory model is enabled, the compiler must use a longer instruction including a 32-bit address to access a data object, whether static or global. This operation requires an additional word. Consequently, the big memory model leads to a larger, and in some cases, slower source code than the equivalent source code written with small memory model enabled.

- Big memory model with far runtime library calls

  Appropriate when code and static data require more than 64KB but less than 1MB of memory for all, but the runtime code. In this model, the runtime code can be more than 1MB away from the rest of the program.

- Huge memory model

  Appropriate when the application requires more than 1MB of memory.

Memory models affect how the StarCore assembler interprets the memory addresses and allocates the memory space. The listing below shows an example.

Listing: Assembler interpretation of memory models

```
; Big memory model(3 16-bit words):
; Interprets address as 32-bit, and allocates maximum space:

move.l    address,d0

; Small memory model (2 16-bit words):

; The < symbol indicates 16-bit space, and saves memory space:

move.l    <address,d0
```

Certain instructions are valid only when used in small memory mode. For example, if < is omitted in the following instruction, an error message occurs:

```
  bmset.w #0001,<address
```

The default memory layout for StarCore architectures is a single linear block divided into code and data areas. C/C++ language programs generate code and data in sections. The compiler places each of these sections in its own continuous space in the memory.

Heap and stack use the same memory representing the default dynamic configuration. In a static configuration, heap and stack use different memory areas. However, you can configure the memory layout for your specific requirements. See *StarCore SC3000 Linker User Guide* for more information.

The compiler uses heap for dynamic allocation of memory. The compiler allocates memory from a global pool for the stack and the heap together. The heap starts at the top of memory, and is allocated in a downward direction toward the stack.

Objects that are dynamically allocated are addressed only with pointers, and not directly. The amount of space that can be allocated to the heap is limited by the amount of available memory in the system.

To make more efficient use of the space allocated to data, you can use the heap to allocate large arrays, instead of defining them as static or global. For example, a definition such as `struct large array1[80];` can be defined using a pointer and the `malloc` function:

```
struct large *array1;

array1 = (struct large *)malloc(80*sizeof(struct large));
```

The compiler allocates an area of memory to the runtime stack, which is used for:

- allocation of local variables
- passing arguments to functions
- saving function return addresses
- saving temporary results

The stack is allocated in the area above the space used for the source code, and grows in an upward direction towards the top of memory. The compiler uses the SP register to manage this stack.

The StarCore architecture includes two stack pointers:

- DSP, used when the processor jumps to service a Debug exception
- ESP, used when the processor is running in Exception mode; default mode at initialization

The compiler makes no assumptions about which stack pointer to use, and uses the pointer for the current processor mode to point to the address at the top of the stack.

When the system is initialized, the stack pointer for the current mode is set by default to the address of the location directly after the code area, as defined in `StackStart` in the linker command file. The actual address of the stack is determined at link time.

The stack pointer for the current processor mode is automatically incremented by the C/C++ environment at the entry to a function. This ensures that sufficient space is reserved for the execution of the function. At the function exit, the stack pointer is decremented, and the stack is restored to its previous size prior to function entry. If your application includes assembly language routines and C/C++ code, you must ensure at the end of each assembly routine that the current stack pointer is restored to its pre-routine entry state.

NOTE

If you change the default memory configuration, make sure to allow sufficient space for the stack to grow. A stack overflow at

runtime, might cause your application to fail. The compiler does not check for stack overflow during compilation or at runtime.

When you compile your application without cross-file optimization, the allocations for each file are assigned to different sections of data memory. At link time these are dispatched to different addresses.

When compiling with cross-file optimization, the compiler uses the same data section for all allocations. If you want to override this and to instruct the compiler to use non-contiguous data blocks, you can edit the machine configuration file to define the exact memory map of the system that you want to use.

Related Tasks

- How to specify memory model

3.1.4 Understanding floating point support in SC3900FP compiler

This topic explains hardware and software floating point support in SC3900FP compiler.

3.1.4.1 Hardware floating point support in SC3900FP compiler

The hardware floating point support is available only in SC3900FP compiler.

The hardware floating point support allows the compiler to perform single precision floating point arithmetic, partially compliant with the IEEE 754. with the following exceptions:

- Only round-to-nearest-even, IEEE default rounding, rounding method is supported.
- De-normalized input is considered by SC3900FP to have zero value, that is single precision numbers are flushed to zero. No de-normalized numbers other than zero are generated as a calculation result.
- No hardware exceptions are automatically generated.

The hardware floating point support is not fully compatible with the SC3900FP software implementation of single floating point arithmetic. This is because the software implementation generates de-normalized numbers other than zero as the calculation results and does not consider de-normalized inputs to have zero value.

The hardware floating point support handles only the single precision floating point arithmetic. For double precision floating point arithmetic, the software implementation is always used, and therefore the SC3900FP de-normalized double precision numbers are not flushed to zero.

3.1.4.2 Software floating point support in SC3900FP compiler

The software floating point support implements both single and double precision floating point arithmetic, partially compliant with IEEE 754, with the following exception:

- Only round-to-nearest-even, IEEE default, rounding method is supported.

3.1.4.2.1 FLUSH_TO_ZERO

NOTE

The `FLUSH_TO_ZERO` option is available only in software floating point library.

This is a boolean configuration item that sets the behavior of un-normalized numbers. When set to true (the default) all un-normalized values are flushed to zero. This leads to better performance, but a smaller dynamic range.

For example, The listing below shows how to disable the `FLUSH_TO_ZERO` option.

Listing: Disabling Flushing to Zero

```
#include <fltmath.h>
. . .
FLUSH_TO_ZERO = 0;
```

Related Tasks

- How to enable support for SC3900FP compiler
- How to disable hardware floating point support in SC3900FP compiler

Related References

- Floating-point characteristics (float.h)
- Floating-Point library interface (fltmath.h)

3.2 General compiler concepts

In this section:

3.2.1 Understanding optimizer

The optimizer converts preprocessed source files into assembly-language output code, applying a range of code transformations that significantly improve the efficiency of the executable program. The optimizer improves code performance in terms of execution time or code size, by producing the output code that is functionally equivalent to the original source code.

The traditional compiler optimizers analyze each source code file individually before the linker links them together. However, for optimal performance, the optimizer must analyze all the source files together. This approach is also known as the cross-file optimization approach. The cross-file optimization approach substantially improves the optimization, and the final optimized code is more efficient than the optimized code generated without cross-file optimization.

The optimizer applies most of these transformations to basic blocks of code. A basic block is a linear sequence of instructions for which there is only one entry point and one exit point. There are no branches in a basic block. In general, bigger basic blocks enable better optimization, as they increase the scope for further optimization.

The optimizer makes the source code take full advantage of the multiple execution units in the StarCore architecture by transforming linear code into parallelized code. The executable programs process instructions in the form of execution sets, with one execution set per cycle. The optimizer can increase the number of instructions in an execution set, so that two or more execution units process instructions in parallel in the same cycle.

- Linear code occupies one execution unit, no matter how many execution units are available. Each execution set consists of one instruction.
- Parallelized code execution sets can contain multiple instructions. If multiple execution units are available, they can execute the execution sets in parallel. Parallelized code executes faster and more efficiently than linear code.

Figure 3-2 illustrates the transformation of linear code into parallelized code.

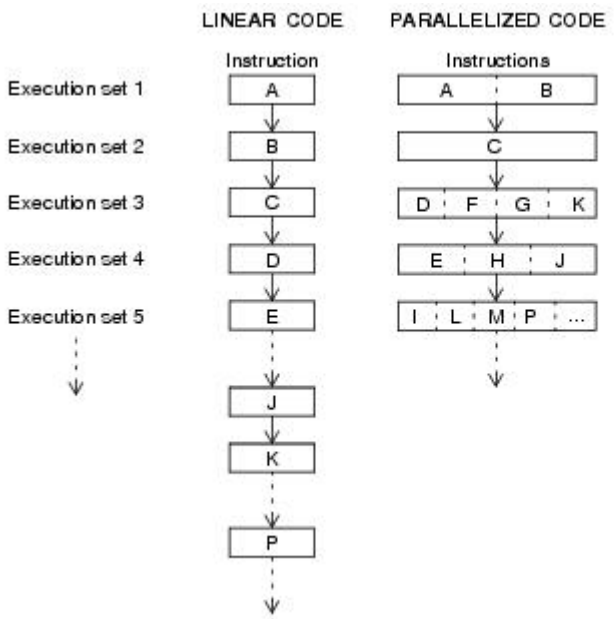

Figure 3-2. Transformation of linear code into parallelized code

The StarCore compiler supports four optimization levels described in Table 3-3.

Table 3-3. Optimization levels

Level	Command-line option	Description	Effects
Level 0	`-O0`	Generates un-optimized code	Compiles fastest
Level 1	`-O1`	Applies target-independent optimizations.This level yields optimized linear code.	Compiles slower than level 0, but generates code that runs faster
Level 2/3(default)	`-O2/-O3`	Applies target-independent optimizations and target-specific optimizations.In addition, the low-level optimizer performs global register allocation. This level yields parallelized code.	Takes advantage of parallel execution units, generating the highest performance code possible without cross-file optimization.Compromises size for speed and generates lesser number of cycles
Level 4	-O4	Performs advanced scalar and loop optimizations. Note that the optimizations enabled by `-O4` are currently experimental, and are not deemed safe in all cases.	Additional optimizations for speed

In addition to the four levels of optimization, the compiler also supports the options for size and cross-file optimization. Table 3-4 describes these additional options.

Table 3-4. Additional optimization options

Option	Description	Benefits
`-Od`	Performs minimum optimization (tracking register content) for relatively small code size; can be specified together with any of the other optimization options except the `-O0` option.	Preserves accuracy of debug information; reduces number of spills in the stack
`-Os`	Performs the specified level of optimization with primary emphasis on reducing the code size; can be specified together with any of the other optimization options except the `-O0` option. When used in conjunction with the `-O3` option, generates the smallest code	Produces smaller optimized assembly code
`-Og`	Performs cross-file optimization; can be specified together with any of the other optimization options except the `-O0` option.	Produces the most efficient results when used with the `-O3` option

Related Tasks

- How to improve performance of the generated code
- How to optimize C/C++ source code

Related Concepts

- Understanding intrinsic functions

3.2.2 Understanding overflow behavior

The SC3900FP compiler has an undefined behavior on signed integer overflow. The result of such an operation does not guarantee the result.

In the case of unsigned integer overflow, the operation always wraps around.

3.2.3 Understanding fractional and integer arithmetic

Fractional arithmetic is important for computation-intensive algorithms, such as digital filters, speech coders, digital control, or other signal-processing tasks. In such algorithms, the compiler interprets the data as fractional values and performs the computations accordingly. However, fractional data types are not supported in native C language. Therefore, the StarCore compiler provides fractional intrinsic functions to support fractional data types. Fractional arithmetic examples include:

- `0.5 * 0.25 -> 0.125`
- `0.625 + 0.25 -> 0.875`
- `0.125 / 0.5 -> 0.25`
- `0.5 >> 1 -> 0.25`

The compiler often uses saturation for signal-processing tasks. This prevents the severe output-signal distortions that might occur because of computation overflows that arise without saturation. You can selectively enable and disable saturation, so that limiting applies only to the final results, and not to the intermediate calculations.

Integer arithmetic is important for controller code, array indexing, address computations, peripheral setup and handling, bit manipulation, and other general-purpose tasks. Integer arithmetic examples include:

- `4 * 3 -> 12`
- `1201 + 79 -> 1280`
- `63 / 9 -> 7`
- `100 << 1 -> 200`

The user application needs to determine whether the compiler interprets the data in a specific memory location as fractional data or integer data. Table 3-5 lists both the interpretations for a 16-bit value, depending upon the location of the binary point.

NOTE

The binary representation in Table 3-5 corresponds to the location of the binary point for the fractional data interpretation. For the integer data interpretation, the binary point is immediately to the right of the LSB.

Table 3-5. Interpretation of 16-bit Data Value

Binary Representation	Hexadecimal Representation	Integer Value (Decimal)	Fractional Value (Decimal)
0.100 0000 0000 0000	0x4000	16384	0.5
0.010 0000 0000 0000	0x2000	8192	0.25
0.001 0000 0000 0000	0x1000	4096	0.125
0.111 0000 0000 0000	0x7000	28672	0.875
0.000 0000 0000 0000	0x0000	0	0.0

Table continues on the next page...

Table 3-5. Interpretation of 16-bit Data Value (continued)

Binary Representation	Hexadecimal Representation	Integer Value (Decimal)	Fractional Value (Decimal)
1.100 0000 0000 0000	0xC000	-16384	-0.5
1.110 0000 0000 0000	0xE000	-8192	-0.25
1.111 0000 0000 0000	0xF000	-4096	-0.125
1.001 0000 0000 0000	0x9000	-28672	-0.875

The following equation shows the relationship between a 16-bit integer and a fractional value:

```
Fractional Value = Integer Value / (215)
```

Table 3-6 lists the fractional and integer interpretation of a 40-bit value, depending upon the location of the binary point.

Table 3-6. Interpretation of 40-bit Data Value

Hexadecimal Representation	40-Bit Integer in Entire Accumulator	16-Bit Integer in MSP (Decimal)	Fractional Value (Decimal)
0x0 4000 0000	1073741824	16384	0.5
0x0 2000 0000	536870912	8192	0.25
0x0 0000 0000	0	0	0.0
0xF C000 0000	-1073741824	-16384	-0.5
0xF E000 0000	-536870912	-8192	-0.25

The following equation shows the relationship between a 40-bit integer and a fractional value:

```
Fractional Value = Integer Value / (231)
```

3.2.3.1 Arithmetic support on StarCore processors

The compiler natively supports these arithmetic operations:

- 40-bit add/subtract with and without saturation
- 32-bit add/subtract without saturation
- 32-bit fractional multiplication without saturation
- 32-bit integer multiplication
- 16-bit fractional multiplication with and without saturation
- 16-bit integer multiplication
- 40-bit logical operations

- 40-bit shift operations
- 32-bit shift operations
- 40-bit comparisons
- 32-bit comparisons
- 8-bit application specific arithmetic
- 16-bit SIMD operations
- 20-bit SIMD operations
- 64-bit add/subtract
- 64-bit shift operations

The type of arithmetic is specified in the instruction syntax, and unless otherwise specified, the operations performed are fractional operations. Instructions performing integer operations include the "i" flag in their syntax. The fractional arithmetic instructions perform a left shift by 1 bit after a multiply operation and may saturate depending on the instruction. The saturation behavior can be extracted from the instruction syntax - the "s" flag marks an always saturating instruction, as opposed to one that does not contain the "s" flag so it never saturates. For instance, mac.s.x designates a signed fractional multiply-accumulate operation that always saturates the results while mac.x refers to a non-saturating multiply-accumulate. For both data types, integer and fractional, and all supported sizes there exist load and store instructions.

Figure 3-3 shows generated assembly instructions for equivalent fractional and integer operations.

Figure 3-3. Assembly Instructions for Equivalent Fractional and Integer Operations

Related Tasks

- How to use fractional data types

3.2.4 Understanding intrinsic functions

The StarCore instruction set includes both integer and fractional instructions. The integer instructions are accessed by the normal C language source code. Since native C language does not support fractional arithmetic, the StarCore compiler supports intrinsic (built-in) functions that map directly to the assembly instructions, and let you access fractional instructions and other architectural primitives.

The listing below shows an example of C source code that does not use intrinsics.

Listing: Sample C source code

```
short SimpleFir0(short *x, short *y) {
    int i;
    short ret;
    ret = 0;
    for(i=0;i<16;i++)
        ret += x[i]*y[i];
    return(ret);
}
```

Figure 3-4. Sample C source code

The listing below shows the assembly code that is generated.

Listing: Generated assembly code

```
;*********************************************************************
;*                                                                   *
;* Function Name:            _SimpleFir0                             *
;* Stack Frame Size:         8 (0 from back end)                     *
;* Calling Convention:       17                                      *
;* Parameter:                x   passed in register r0               *
;* Parameter:                y   passed in register r1               *
;*                                                                   *
;* Returned Value:           ret_SimpleFir0   passed in register r0  *
;*                                                                   *
;*********************************************************************
                GLOBAL                _SimpleFir0
                ALIGN   16
_SimpleFir0        TYPE      func OPT_SPEED
                SIZE _SimpleFir0,F_SimpleFir0_end-_SimpleFir0,16
;PRAGMA stack_effect _SimpleFir0,8
```

```
[
                adda.lin        #8,sp                   ;[1,1]
                eor.x           d0,d0,d0                ;[5,1]
                tfra.l          #4,r16                  ;[0,0]
                adda.lin        #<2,r1,r2               ;[0,1]
                eor.x           d1,d1,d1                ;[7,1]
                eor.x           d2,d2,d2                ;[7,1]
                eor.x           d3,d3,d3                ;[7,1]
]
DW_2 TYPE debugsymbol
[
                anda            r1,r1,r8                ;[1,1]
                anda            r0,r0,r1                ;[1,1]
                adda.lin        #<2,r0,r3               ;[0,1]
]
[
                ld.w            (r1)+r16,d5             ;[7,1] 0%=0 [0]
                ld.w            (r8)+r16,d4             ;[7,1] 0%=0 [0]
                adda.lin        #<2,r2,r4               ;[0,0]
]
[
                mac.i.x         d4.l,d5.l,d0            ;[7,1] 1%=0 [0]
                ld.w            (r2)+r16,d4             ;[7,1] 1%=0 [0]
                adda.lin        #<2,r3,r5               ;[0,0]
                ld.w            (r3)+r16,d5             ;[7,1] 1%=0 [0]
]
[
                doen.3          #3                      ;[0,1] @II4
                mac.i.x         d4.l,d5.l,d1            ;[7,1] 2%=0 [0]
                adda.lin        #<2,r4,r6               ;[0,0]
                adda.lin        #<2,r5,r7               ;[0,0]
]
[
                ld.w            (r4)+r16,d4             ;[7,1] 2%=0 [0]
                ld.w            (r5)+r16,d5             ;[7,1] 2%=0 [0]
```

```
]
[
                mac.i.x         d4.l,d5.l,d2          ;[7,1] 3%=0 [0]
                ld.w            (r7)+r16,d5           ;[7,1] 3%=0 [0]
                ld.w            (r6)+r16,d4           ;[7,1] 3%=0 [0]
]
                LOOPSTART3
L5
[
                mac.i.x         d4.l,d5.l,d3          ;[0,1] 4%=1 [0]
                ld.w            (r1)+r16,d5           ;[7,1] 0%=0 [1]
                ld.w            (r8)+r16,d4           ;[7,1] 0%=0 [1]
]
[
                mac.i.x         d4.l,d5.l,d0          ;[7,1] 1%=0 [1]
                ld.w            (r3)+r16,d5           ;[7,1] 1%=0 [1]
                ld.w            (r2)+r16,d4           ;[7,1] 1%=0 [1]
]
[
                mac.i.x         d4.l,d5.l,d1          ;[7,1] 2%=0 [1]
                ld.w            (r5)+r16,d5           ;[7,1] 2%=0 [1]
                ld.w            (r4)+r16,d4           ;[7,1] 2%=0 [1]
]
[
                mac.i.x         d4.l,d5.l,d2          ;[7,1] 3%=0 [1]
                ld.w            (r7)+r16,d5           ;[7,1] 3%=0 [1]
                ld.w            (r6)+r16,d4           ;[7,1] 3%=0 [1]
]
                LOOPEND3
[
                mac.i.x         d4.l,d5.l,d3          ;[0,1] 4%=1 [1]
                adda.lin        #-8,sp                ;[8,1]
]
DW_13 TYPE debugsymbol
                add.x           d0,d3,d0              ;[7,1]
```

```
                add.x           d0,d2,d0                ;[7,1]
                add.x           d0,d1,d0                ;[7,1]
                move.l          d0,r0                   ;[7,1]
                sxta.w.l        r0,r0                   ;[0,0]
                rts                                     ;[8,1]
                GLOBAL                  F_SimpleFir0_end
F_SimpleFir0_end
FuncEnd_SimpleFir0
TextEnd_test_manual
                ENDSEC
```

The listing below shows an example of C source code that uses intrinsics.

Listing: Sample C source code

```
#include <prototype_sc3900.h>
short SimpleFir1( short *x, short *y) {
    int i;
    int ret;
    ret = 0;
    for(i=0;i<16;i++)
        ret=__l_mac_x_hh(ret,__l_put_msb(x[i]),__l_put_msb(y[i]));
    return(__l_get_msb(ret));
}
```

The listing below shows the assembly code that is generated.

Listing: Generated assembly code

```
;*********************************************************************
;*                                                                   *
;* Function Name:              _SimpleFir1                           *
;* Stack Frame Size:           8 (0 from back end)                   *
;* Calling Convention:         17                                    *
;* Parameter:                  x   passed in register r0             *
;* Parameter:                  y   passed in register r1             *
;*                                                                   *
;* Returned Value:             ret_SimpleFir1   passed in register r0 *
;*                                                                   *
```

```
;*********************************************************************
                GLOBAL                  _SimpleFir1
                ALIGN   16
_SimpleFir1         TYPE      func OPT_SPEED
                SIZE _SimpleFir1,F_SimpleFir1_end-_SimpleFir1,16
;PRAGMA stack_effect _SimpleFir1,8
[
                tfra.l          #<4,r4              ;[0,1]
                adda.lin        #<2,r1,r2           ;[0,1]
                adda.lin        #<2,r0,r3           ;[0,1]
                eor.x           d0,d0,d0            ;[5,1]
]
[
                doen.3          r4                  ;[0,1]
                adda.lin        #<2,r2,r4           ;[0,0]
                adda.lin        #<2,r3,r5           ;[0,0]
]
[
                adda.lin        #8,sp               ;[2,1]
                anda            r1,r1,r8            ;[2,1]
                anda            r0,r0,r1            ;[2,1]
]
DW_4 TYPE debugsymbol
 [
                adda.lin        #<2,r4,r6           ;[0,0]
                adda.lin        #<2,r5,r7           ;[0,0]
                tfra.l          #4,r0               ;[0,0]
]
                LOOPSTART3
L5
[
                ld.f            (r1)+r0,d2          ;[7,1]
                ld.f            (r8)+r0,d1          ;[7,1]
]
[
```

```
                mac.x           d2.h,d1.h,d0        ;[7,1]
                ld.f            (r3)+r0,d4          ;[7,1]
                ld.f            (r2)+r0,d3          ;[7,1]
]
[
                ld.f            (r5)+r0,d1          ;[7,1]
                ld.f            (r4)+r0,d2          ;[7,1]
]
[
                mac.x           d4.h,d3.h,d0        ;[7,1]
                ld.f            (r7)+r0,d4          ;[7,1]
                ld.f            (r6)+r0,d3          ;[7,1]
]
                mac.x           d1.h,d2.h,d0        ;[7,1]
                mac.x           d4.h,d3.h,d0        ;[0,1]
                LOOPEND3
                st.l            d0,(sp-8)           ;[0,1]
                ld.w            (sp-8),d0           ;[8,1]
[
                move.l          d0,r0               ;[8,1]
                adda.lin        #-8,sp              ;[8,1]
]
DW_14 TYPE debugsymbol
                sxta.w.l        r0,r0               ;[0,0]
                rts                                 ;[8,1]
                GLOBAL                  F_SimpleFir1_end
F_SimpleFir1_end
FuncEnd_SimpleFir1
TextEnd_test_manual
                ENDSEC
```

The listing above shows that when intrinsic functions are used, the compiler generates different assembly instructions. The data is loaded using fractional load instructions (`move.f`) and uses fractional arithmetic instructions (`mac`). The `mac` instruction performs a left shift after the multiply instruction and saturates after the addition, if necessary.

Related Tasks

- How to improve performance of the generated code
- How to optimize C/C++ source code

3.2.5 Understanding cw_assert function

The `cw_assert` function lets user indicate certain source code properties. The information so provided might be helpful to the compiler for optimizing the source code.

The `cw_assert` function is inserted in the application code as a function call with a condition as parameter. It has two different behaviors, depending on the optimization level:

- With no optimizations (`O0`), the `cw_assert` function is transformed into a library function call. The `cw_assert` implementation checks the condition given as parameter, and if false, it generates an error. This helps application developers validate the information passed to `cw_assert`.
- For all other optimization levels, the `cw_assert` function lets user indicate certain source code properties as a hint for compiler optimizations. The compiler will interpret the condition, consider it true, and use it as input for its algorithms. It will never check the legality of the logical expression.

The information so provided might be helpful to the compiler for optimizing the source code. The support is restricted to establishing range (minimum, maximum values) and modulo (or alignment) for user variables. The condition can establish order relations (`>,>=,<,<=,==`) between C variables and constants, or modulo properties (`var&0x01..1 == 0 OR var%const == 0`). If the `cw_assert` condition is not recognized by the compiler and cannot be used in the two scopes above, it is ignored, eliminated and does not reach the final code.

As an example, `cw_assert((p&0x7)==0)` tells the compiler that `p` is aligned at `8`, `cw_assert((var %2)==0 && var>10)` tells the compiler that var is an even number > 10, but `cw_assert((var1+var2)<10)` is not understood by the compiler.

3.2.5.1 Range analysis and loop optimizations

In *range analysis*, the compiler assesses the dynamics of a given variable at a given location. This knowledge might show that it is safe to substitute simpler or less-expensive code. The latest range analyzer modules are more accurate and can use functions such as `cw_assert` to pass information to the compiler.

The position of function `cw_assert` - in code - means that passed information is both context sensitive and flow sensitive. For example, it is possible to specify via `cw_assert` that a dynamic loop must iterate a given number of times, and that the iteration count must be a multiple of a particular constant. This information helps the compiler determine whether it is possible to unroll the loop.

Similarly, the function could pass alignment conditions in a context-dependant manner. Through `cw_assert`, for example, the compiler could know the current value of a given pointer, if 8-byte aligned - before entering a loop.

3.2.5.2 Forcing alignment

Another use for a dedicated pragma or function `cw_assert` is forcing or indicating alignment.

One way to inform the compiler how a pointer or array is aligned, is to use . However, note that this has wide visibility, so it is not very accurate.

- For pointers, the syntax is:
- For arrays, the syntax omits the asterisk:

3.2.5.3 __attribute__((aligned(<n>)))

One way to inform the compiler how a pointer or array is aligned, is to use `__attribute__((aligned(<n>)))`. However, note that this attribute has wide visibility, so it is not very accurate.

- For pointers, the syntax is:

```
int *__attribute__((aligned(8))) a;
```

- For arrays, the syntax omits the asterisk:

```
int __attribute__((aligned(8))) a[100];
```

3.2.5.4 Using cw_assert for alignment

The `cw_assert` function provides another way to indicate pointer or array alignment: useful information for extending the scope of packing transformations. In the listing below, `cw_assert` indicates that the `pin` pointer is 8-aligned (pin%8=0).

Listing: Basic cw_assert packing example

```
...
short fct_assert_inptr1(int* pin)
__attribute__ ((noinline))

{

 int i, j, kiff, index, var;
 cw_assert(((int)pin%8)==0);
 var=4;
    j=0;

 for(i=0; i<8; i+=2) {
 pin[i+var] = j;
 pin[i+var+1] = j+1;
 j+=2;
 }
 return(pin[0]+var);
}

...
```

Knowing this alignment, and knowing that the initial values of `i` and `var` are 0 and 4, the compiler can use two-moves for the two accesses in the loop body. (For this example, `__attribute__((aligned(<n>)))` would be equivalent.)

NOTE

The (int) cast is mandatory as modulo arithmetic does not apply to pointers. The front-end rejects such cases.

The listing below is a complex variation of the code in shown in the above listing.

Listing: Complex cw_assert packing example

```
...
short fct_assert_inptr1(int var, int* pin)
__attribute__ ((noinline))

{

 int i, j, kiff, index;
 cw_assert(((int)pin%8)==0);
 cw_assert(var%8)==0);
    j=0;

 for(i=0; i<8; i+=2) {
 pin[i+var] = j;
 pin[i+var+1] = j+1;
 j+=2;
 }
 return(pin[0]+var);
}

...
```

Combining a `cw_assert` for the `pin` pointer on one side, with a `cw_assert` for `var` on the other side, enables the compiler to determine that alignment is valid for multiple moves. This mechanism makes it possible to extend the scope of this transformation. This is especially useful for library functions that receive pointers and offsets as inputs.

NOTE

It is possible to merge the two `cw_assert` lines in the code listed above into one. For example, `cw_assert(((int)pin%8)==0 && var %8==0);`

3.2.5.5 Cast simplification in intermediate language

The compiler often promotes types when integral types are not used. This creates several temporary variables in the intermediate language, variables that make code hard to analyze and add several casts. Furthermore, it is not always easy to decrypt the created code patterns.

But type promotion is not necessary if there is no variable overflow. *Cast simplification* is removing type promotion wherever possible, so that the system performs instructions on the initial types of variables. The listing below shows source code for an example.

Listing: Range analysis: cast simplification source code

```
#include <prototype.h>

int alpha(short a)

{

    cw_assert(-10<=a && a<=10);

    a = a+1:

    if (a<0)

        a = -a;

    return a;

}
```

Without benefit of the range analyzer, the compiler promotes the type of variable `a` in both the addition `a = a+1` and the assignment `a = -a`. This prevents detection of the absolute value, resulting in the below listed code.

Listing: Range analysis: code with type promotion

```
    inc d0
    tstge d0
```

```
    iff sxt.w d0,d0
  [
    iff neg d0
    ifa rtsd
  ]
    sxt.w d0,d0
```

But the range analyzer knows that the `cw_assert()` instruction prevents wraparound in either `a = a+1` or `a = -a`. This makes absolute value detection possible in the intermediate language, shortening code to that of the code listed below.

Listing: Range analysis: code with cast simplification

```
    inc d0
  [
    abs d0
    rts
  ]
```

3.2.6 Understanding emulation library

For an application that runs on a host other than StarCore 3900FP DSP, the StarCore emulation library lets the applications use StarCore intrinsics.

Following table lists platforms and tools that the emulation library currently supports and how the library is integrated.

Table 3-7. Platforms and Tools Supported by Emulation Library

Supported Platform and Tools	Integrated As
Windows - Visual Studio and CL compiler	`prototype.lib`, `prototype.dll`, `prototype.h` for SC3850, `prototype_sc3900.h` for SC3900FP
Linux (Red Hat 5) - GCC	`libprototype.so.1.0`

The code matches the StarCore intrinsic with the equivalent emulation code in the Emulation library, compiles and executes without any error, and generates the output similar to that generated in StarCore environment.

The emulation library is available at the following location:

CWInstallDir /SC/StarCore_Support/compiler/library

where, *CWInstallDir* is the StarCore installation directory.

Note

- You do not need to modify the code of your application to use the emulation library.

Related Task

- How to use emulation library

Related Reference

- Intrinsics supported in emulation library

3.2.7 Understanding MEX-library

The StarCore MEX-library enables emulations of the StarCore SC3900FP intrinsics in the MATLAB® environment.

The MEX-library is built upon StarCore compiler's prototype library by using the external interfaces from MATLAB® environment.

However, note that the MEX-library does not introduce new prototypes in the StarCore compiler's prototype library.

The StarCore compiler provides the MEX-library as a zip archive, `sc3900_mex_library.zip`, along with `prototype_sc3900.dll` in the `install-dir/library` sub-directory.

The zip archive contains `.mexw32` files, each of which corresponds to one intrinsic and bears the name of the intrinsic prefixed by `sc3900`. For instance, the counterpart of the 40-bit addition intrinsic `__add_x` is `sc3900__add_x` in the MATLAB® library, and its corresponding file in the library is `sc3900__add_x.mexw32`.

NOTE

Versions of MATLAB® prior to 7.1 do not recognize MEX-files that have the `.mexw32` extension.

NOTE

Not all intrinsics supported by StarCore compiler's prototype library are available in the MEX-library. This is because not all of the intrinsics are relevant in the MATLAB® environment. For example, the prototypes used for memory transfer or the prototypes used for interrupt enabling/disabling.

NOTE

While MATLAB® is centered on matrix operations, the MEX-library functions currently operate only on single elements. Matrix operations are currently not supported. Therefore, if you use matrix operands, then only the first element is taken into consideration by the function.

3.2.7.1 Data types and operations

When calling a library function with immediate values, the MATLAB® default data type (double) is used, meaning that no explicit cast is necessary. Values are returned from functions as double.

The MEX-library defines the Word40 as struct. Setting a 40-bit value is performed with the `__x_set` intrinsic, `sc3900__x_set` in Matlab.

For example, in order to add two 40-bit values without saturation, use the following:

```
>> X_a = sc3900__add_x( sc3900__x_set( 1, 65535 ), sc3900__x_set( 2, 0 ))
```

The 40-bit result will be displayed as follows:

```
ans =
      ext: 3
      body: 65535
```

Notes

- The `install-dir` directory refers to the default location where StarCore compiler directories are stored.

Related Task

- How to use MEX-library in MATLAB® environment

Related Reference

- Intrinsics supported in emulation and MEX library

3.2.8 Understanding modulo addressing

When a modulo operation causes memory addresses to be accessed in a wraparound pattern, the CodeWarrior for StarCore compiler tries to recognize the modulo behavior.

Consider the following function:

Listing: Example - Modulo Addressing

```
int Arr[30];
void test_modulo_addressing()

{

   int i;

   int j;

   for (i = 10; i < 30; ++i)

   {

      j = i % 20;

      Arr[j] = i;

   }

}
```

After `Arr[19]` is accessed, the next iteration of the loop accesses `Arr[0]` instead of `Arr[20]` because of the modulo operation.

When the optimization level is 1 or greater and the compiler is optimizing for speed, it recognizes the modulo behavior and rewrite the function, as the following code shows:

Listing: Code Rewritten by Compiler after Recognizing Modulo behavior

```
int Arr[30];
void test_modulo_addressing()

{

   int i;

   int j;


   __ModuloBuffer buf = __ConstructModuloBuffer(&Arr[0], 20);

   int* pi = &Arr[10];

   for (i = 10; i < 30; ++i)

   {

        *pi = i;

        pi = __AddWithModulo(pi, 1, buf);

   }

   buf = __DestructModuloBuffer(buf);

 }
```

The `__ConstructModuloBuffer` call sets up a modulo buffer with a base address of `&Arr[0]` and a size of 20 elements.

The `__AddWithModulo` call produces a sum of its first two inputs within the modulo buffer that is its third input. In this case it increments the `pi` pointer to the next element of the `Arr` array. When the pointer reaches the end of the modulo buffer, it wraps around to the beginning. So, the call works the same as:

```
pi = &Arr[0] + ((pi - &Arr[0]) + 1) % 20;
```

The difference is that the `__AddWithModulo` call is a strong hint that the compiler should try to speed up the execution by utilizing the available modulo addressing hardware.

Finally, the `__DestructModuloBuffer` call releases the modulo buffer.

NOTE

All of these calls are merely an illustration of the compiler's internal representation of the function. You should not try to call them directly.

The backend of the compiler finishes the compilation by utilizing the special modulo addressing mode of the `SC3900 AGU`. The final assembly code for the function is:

Listing: Assembly Code

```
                GLOBAL                  _test_modulo_addressing
                ALIGN    32

_test_modulo_addressing                 TYPE       func OPT_SPEED

                SIZE
_test_modulo_addressing,F_test_modulo_addressing_end-
_test_modulo_addressing,32

;PRAGMA stack_effect _test_modulo_addressing,0

[

                tfra.l          #_Arr,r1                ;[0,1]

                doen.0          #+20                    ;[0,1]

                tfra.l          #+80,r24                ;[0,1]

                tfr.x           #<+10,d2                ;[8,1]

]

                adda.lin        #+40,r1,r0              ;[0,1]

[

                bmseta          #+8,mctl.l              ;[0,0]

                ora             r1,r1,r8                ;[0,0]

]

                LOOPSTART0

L2

[
```

```
                st.l            d2,(r0)+                ;[11,1]
                add.x           #<+1,d2,d2              ;[0,1]
]
                LOOPEND0
 [
                bmclra          #+15,mctl.l             ;[0,0]
                rts                                     ;[13,1]
]
               GLOBAL                F_test_modulo_addressing_end
```

3.2.9 Understanding predication

The SC3900FP core boasts 6 predicates (p0-p5) that are taken advantage of to boost performance. The goal is to improve execution speed and it is achieved by reducing the number of change of flow instructions executed at run time, by giving other optimizations the opportunity to be applied on a larger window of instructions and by enabling instruction pipelining and better scheduling due to the reduced dependencies. This enablement is derived from the ability to generate up to six predicates to condition the executed instructions.

During if-conversion, conditioned change of flow instructions are removed and the instructions jumped over are instead conditioned with a condition opposite to the one the change of flow was executed on. Having more conditions available for this operation allows the removal of more change of flow instructions in more contexts.

The conditioned section of the flow graph will be concatenated to the previous section due to the removal of the change of flow which will enable more optimizations.

Listing: C Code Example

```
if ((a == 0) || (a == 1))
  {
   d += b;
  }
if (b == 0)
  {
   e += 2;
  }
if (c > 10)
```

```
   {
     f --;
   }
Generated code:
 [
                cmp.eq.l        #<+1,d47,p0:p1      ;[15,1]
                cmp.eq.l        #<+0,d47,p0:p1      ;[15,1]
                ld.l            (r1),r2             ;[19,1]
                ld.l            (r10),r4            ;[23,1]
                tfra.l          #_e,r21             ;[0,1]
 ]
                ld.l            (r21),d17           ;[0,1]
                cmpa.ne         #<+0,r2,p2:p3       ;[19,1]
 [
    IF.p0       ld.l            (r3),d12            ;[17,1]
    IF.p0       ld.l            (r1),d41            ;[17,1]
    IF.p3       add.x           #<+2,d17,d17        ;[21,1]
 ]
 [
    IF.p0       add.x           d41,d12,d53         ;[17,1]
    IF.p3       st.l            d17,(r21)           ;[21,1]
 ]
 [
    IF.p0       st.l            d53,(r3)            ;[17,1]
                cmpa.le         #<+10,r4,p0:p1      ;[23,1]

]
    IF.p1       sub.x           #<+1,d61,d61        ;[25,1]
    IF.p1       st.l            d61,(r17)           ;[25,1]
```

3.2.10 Understanding code reordering

Code reordering is a transformation performed by the compiler in order to achieve better code. The objective is to reduce the execution time by reducing the number jumps taken at executin time. This is obtained by moving code that is unlikely to execute. The compiler needs user input in order to make this transformation and the user may help it by using the `likely` and `unlikely` macros. As a result, the compiler would know how to reorder the code in such a way that jumps are avoided (in the majority of cases).

The consequence of performing code reordering is that the instructions are rearranged so that the most likely blocks to be executed are grouped in a succession (thus not needing jumps) and the rest of the unlikely code is put at the end of functions. This has another advantage, the code which is unlikely to be executed may not even be loaded from memory into cache since it is located after the end of the function thus reducing unnecessary memory reading. An add-on to this optimization is the splitting of the function in two different sections. The `likely` code remains in place but the `unlikely` one is moved to another section. For example,

Listing: Code Reordering Example

```
int test_assert(int *v, int n)
{

        int i;

        int sum = 0;

        int cnt = 0;

        for (i = 0; i < n; ++i)

         {

            cnt += i;

            if (v[i] % 2)

             {

               myexit(i);

             }

            sum += v[i];

          }

       return sum - cnt;

}
```

For the previous snippet of C code, the generated code looks like:

Listing: Generated Code

```
StackOffset__test_assert EQU 0;  at _test_assert sp = 0
StackOffset_DW_2         EQU 12; at DW_2 sp = 12

StackOffset_DW_18        EQU 0; at DW_18 sp = 0
```

```
*;*****************************************************************;*
*;* Function Name:          _test_assert                            *;*
*;* Stack Frame Size:       56 (0 from back end)                    *;*
*;* Calling Convention:     1                                       *;*
*;* Parameter:              v   passed in register r0               *;*
*;* Parameter:              n   passed in register r1               *;*
*;* Returned Value:         ret_test_assert   passed in register r0 *;*
*;*****************************************************************;*
GLOBAL              _test_assert
ALIGN               32
_test_assert        TYPE
func OPT_SPEED      SIZE
_test_assert,F_test_assert_end-_test_assert,32;
PRAGMA stack_effect _test_assert,56
 [
                cmpa.le       #+0,r1,p5           ;[0,0]
                push.4x       d28:d29:d30:d31     ;[0,0]
                push.4l       r28:r29:r30:r31     ;[0,0]
                eor.x         d30,d30,d30         ;[13,1]
 ]
DW_2 TYPE debugsymbol
 [
                eora          r31,r31,r31         ;[14,1]
                eora          r29,r29,r29         ;[15,1]
                anda          r0,r0,r28           ;[0,0];FixUp Code B1
    IF.p5       bra           L3                  ;[0,0]
 ]
                anda          r1,r1,r30           ;[0,0];FixUp Code
L2
 [
                ld.l          (r28),d37           ;[0,1]
                adda.lin      r29,r31,r31         ;[17,1]
                anda          r29,r29,r0          ;[20,1] B3
 ]
                lsh.rgt.l     #<+31,d37,d44       ;[0,1]
```

```
                add.x           d44,d37,d17          ;[0,1]
                ash.rgt.l       #<+1,d17,d50         ;[0,1]
                ash.lft.x       #<+1,d50,d16         ;[0,1]
                sub.x           d16,d37,d46          ;[18,1]
                cmp.eq.l        #<+0,d46,p0:p1       ;[18,1]
    IF.p0       jmp             L4                   ;[18,1]
                jsr             _myexit              ;[20,1]
                ld.l            (r28),d37            ;[0,1]
L4
 [
                deca.ne         r30,p5               ;[0,0]
                add.x           d37,d30,d30          ;[22,1]
                adda.lin        #<+1,r29,r29         ;[15,1]
                adda.lin        #<+4,r28,r28         ;[0,1]
 ]
    IF.p5       bra             L2                   ;[0,0]
L3
                move.l          d30,r17              ;[24,1]
 [
                suba.lin        r31,r17,r0           ;[24,1]
                pop.4l          r28:r29:r30:r31      ;[0,0]
                pop.4x          d28:d29:d30:d31      ;[0,0]
 ]
DW_18 TYPE debugsymbol
                rts                                  ;[25,1]
GLOBAL    F_test_assert_end
F_test_assert_end
FuncEnd_test_assert
```

Note that the `IF.p0 jmp L4` instruction always executes, when the condition inside the `if` is false. If the condition is false most of the time, then the jump adds an unnecessary delay during execution. The solution to this problem is to use the `unlikely` macro and mark the condition as seldomly true, so that the compiler can optimize the code accordingly.

The `if` statement modified to use `unlikely` looks like:

```
  if (unlikely(v[i] % 2))
```

The generated code looks like the code listed below:

Listing: Generated Code

```
SECTION .text LOCAL
SECFLAGS ALLOC,NOWRITE,EXECINSTR

SECTYPE PROGBITS

TextStart_assert

StackOffset__test_assert        EQU  0  ; at _test_assert sp = 0

StackOffset_DW_2                EQU  12 ; at DW_2 sp = 12

StackOffset_DW_16               EQU  0  ; at DW_16 sp = 0

*;*****************************************************************;*

*;* Function Name:          _test_assert                          *;*

*;* Stack Frame Size:       48 (0 from back end)                  *;*

*;* Calling Convention:     1                                     *;*

*;* Parameter:              v   passed in register r0             *;*

*;* Parameter:              n   passed in register r1             *;*

*;* Returned Value:         ret_test_assert  passed in register r0 *;*

*;*****************************************************************;*

GLOBAL            _test_assert

ALIGN             32

_test_assert      TYPE

func OPT_SPEED

SIZE _test_assert,F_test_assert_end-_test_assert,32

;PRAGMA stack_effect _test_assert,48

 [

                cmpa.le       #+0,r1,p5           ;[0,0]

                push.4x       d28:d29:d30:d31     ;[0,0]

                push.4l       r28:r29:r30:r31     ;[0,0]

                eor.x         d30,d30,d30         ;[13,1]

 ]

DW_2 TYPE debugsymbol

 [

                eora          r31,r31,r31         ;[14,1]

                eora          r29,r29,r29         ;[15,1]

                anda          r0,r0,r28           ;[0,0];FixUp Code B1

    IF.p5       bra           L3                  ;[0,0]

 ]
```

```
                anda            r1,r1,r30           ;[0,0];FixUp Code
L2
 [
                ld.l            (r28),d37           ;[0,1]
                adda.lin        r29,r31,r31         ;[17,1]
                anda            r29,r29,r0          ;[20,1] B3
 ]
                lsh.rgt.l       #<+31,d37,d44       ;[0,1]
                add.x           d44,d37,d17         ;[0,1]
                ash.rgt.l       #<+1,d17,d50        ;[0,1]
                ash.lft.x       #<+1,d50,d16        ;[0,1]
                sub.x           d16,d37,d46         ;[18,1]
                cmp.eq.l        #<+0,d46,p0:p1      ;[18,1]
    IF.p1       jmp.nobtb       assertPF0           ;[18,1]
L4
 [
                deca.ne         r30,p5              ;[0,0]
                add.x           d37,d30,d30         ;[22,1]
                adda.lin        #<+1,r29,r29        ;[15,1]
                adda.lin        #<+4,r28,r28        ;[0,1]
 ]
    IF.p5       bra             L2                  ;[0,0]
L3
                move.l          d30,r17             ;[24,1]
 [
                suba.lin        r31,r17,r0          ;[24,1]
                pop.4l          r28:r29:r30:r31     ;[0,0]
                pop.4x          d28:d29:d30:d31     ;[0,0]
 ]
DW_16 TYPE debugsymbol
                rts                                 ;[25,1]
      GLOBAL
F_test_assert_end
F_test_assert_end
FuncEnd_test_assert
```

```
        ENDSEC

SECTION .unlikely LOCAL

SECFLAGS ALLOC,NOWRITE,EXECINSTR

SECTYPE PROGBITS

StackOffset_assert_test_assertassertPF0         EQU     0    ; at
assert_test_assertassertPF0 sp = 0

;*****************************************************************;*

*;* Function Name:         assert_test_assertassertPF0              *;*

*;* Stack Frame Size:      8 (0 from back end)                      *;*

*;* Calling Convention:    3                                        *;*

*;*****************************************************************;*

ALIGN   32

assert_test_assertassertPF0  TYPE   func OPT_SPEED

SIZE assert_test_assertassertPF0,Fassert_test_assertassertPF0_end-

assert_test_assertassertPF0,32

;PRAGMA stack_effect assert_test_assertassertPF0,8

assertPF0

                jsr           _myexit              ;[20,1]

                ld.l          (r28),d37            ;[0,1]

                jmp           L4                   ;[0,0]

Fassert_test_assertassertPF0_end

FuncEnd_assert_test_assertassertPF0

ENDSEC
```

Note that the `unlikely` code is moved in a separate function in a separate section, the `.unlikely` section. The initial conditioned jump has changed from `IF.p0 jmp L4` to `IF.p1 jmp.nobtbassertPF0` so the jump is taken only when the condition is true, which happens seldomly. Any other `unlikely` portions of code are included in the `.unlikely` section.

Related Tasks

- How to enable code reordering

3.2.11 Understanding function inlining

The inline process in the SC3900 compiler is controlled by the following option:

Option	Description
inline	Specifies the inline options.

The following table lists and describes the parameters for inline option:

Table 3-8. Inline option keywords

Keyword	Command-line option	Description
never	-inline -never	Disables all inlining including always_inline functions.
none | off	-inline -none | off	Turns off the inlining for inline functions.
on | smart	-inline -on | smart	Turns on the inlining for inline functions (default).
auto	-inline -auto	Auto-inlines the small functions (without inline explicitly specified) (by default, activated starting with O2).
noauto	-inline -noauto	Disables the auto-inline.
all	-inline -all	Turns on the aggressive inlining. Same as inline, on and auto functions.

The group of functions that cannot be inlined by the frontend contains:

- Asm functions
- Functions with __attribute__((noinline))
- Functions with side effects in return/arguments (constructors, destructors etc)

In order to guarantee that a function is always inlined (if the compiler can establish the legality), __attribute__((always_inline)) attribute must be used. Note that the restrictions related to recursivity are considered.

The frontend auto-inlines functions without __attribute__((always_inline)) attribute in O2, O3 and O4 optimizations.

In order to be considered for inlining, a function without the attributes __attribute__((always_inline)) or __attribute__((inline)) must pass the inline constraints:

- The caller and the callee have the same optimizations levels

Case 1:

- The function is not part of the never inline group
- The function is used exactly ones
- The estimated size (in bytes) of the function is smaller than 650 (or the value set by #pragma ipa_inline_max_auto_size)

Case 2:

- The function is not part of the never inline group
- The estimated size (in bytes) of the function is smaller than 300 (or 5 times the value set by `#pragma inline_max_auto_size`)

All caller/callee function pairs considered for inlining with the exceptions of the ones with `__attribute__((always_inline))` must respect the following conditions:

- After inline, the estimated size of the caller is either <= 2000 or smaller 4*(initial size)
- After inline, the estimated size of the caller is either <= 10000 or smaller 5/4*(initial size)

`#pragma aggressive_inline on` bypass these conditions.

There is no limit for the depth of inlining, with the exception of recursivity, where restrictions apply.

During size optimizations, a function is inlined if the compiler can establish that after the inline the overall size of the call is not increased:

- Function called exactly once.
- Size of call is higher than the size added after the inline by local parameters.

Related Tasks

- How to inline or prevent inlining of function

Related References

- #pragma inline_max_auto_size
- #pragma ipa_inline_max_auto_size

3.3 Pragmas and attributes concepts

Compiler pragmas follow this general syntax:

```
#pragma pragma-name [argument(s)]
```

Each pragma must fit on one line. One or more of the arguments may be optional; commas must delimit arguments.

Each pragma or attribute applies only to a specific context, so you must place pragmas or attributes accordingly:

- Pragmas and attributes that apply to functions - place the pragmas only in the scope of the function, after the opening brace. The attributes are defined before the opening brace.
- Pragmas and attributes that apply to statements - place the pragmas immediately before the relevant statement, or immediately before any comment lines that precede the statement.
- Pragmas and attributes that apply to variables - place the pragmas after the object definition, or after any comment lines that follow that definition. If a pragma refers to objects, the objects must be defined explicitly.

  Place the attributes between the type name and the variable name.

- Other pragmas and attributes - place them according to their individual requirements.

Related Task

- Function pragmas and attributes tasks
- Statement pragmas tasks
- Other pragmas and attributes tasks

Related Reference

- Function pragmas and attributes
- Statement pragmas and attributes
- Other pragmas and attributes

Chapter 4
References

This section consists of reference information to which you might need to refer, to accomplish compiler tasks.

- Command-line options
- Pragmas and attributes
- Runtime libraries
- Calling conventions
- Predefined Macros
- C++ specific features
- Intrinsics supported in emulation and MEX library

4.1 Command-line options

This appendix lists the command-line options and flags available with StarCore compiler.

In this appendix:

- Shell behavior control options
- Application file options
- Pre-processing control options
- Output file extension options
- C language options
- Optimization and code options
- Shell passthrough options
- File and message output options
- Hardware configuration options
- Library options
- Low level optimizer options

- Compiler front-end warning messages
- Warning index values

4.1.1 Shell behavior control options

Table 4-1 lists the shell behavior control options.

Table 4-1. Shell behavior control options

Option	Description
`-c`	Compiles and assembles only; does not invoke the linker
`-cfe`	Stops after Front End and high-level optimization phase.
`-E [file]`	Stops after preprocessing source files; removes the comments
`-env <path>`	Define the path of the compiler; given path must be the root of the compiler directory; overrides the `SC100_HOME` environment variable
`-F file`	Reads additional shell options from the specified file; appends them to command line
`-h or none`	Displays the shell help page, invocation syntax, and available options
`-S`	Stops after compilation; does not invoke the assembler

4.1.2 Application file options

You use application file options in an application configuration file. Table 4-2 lists the application file options available.

Table 4-2. Application File Options

Option	Default Value	Description
`Allconst_To_Rom`	FALSE	If TRUE, stores in ROM all symbols that the high-level optimizer finds constant.
`Const_To_Rom`	TRUE	If FALSE, does *not* place constants in ROM.
`Init_To_Rom`	FALSE	If TRUE, permits placement of an array initializer in ROM.
String_To_Rom	FALSE	If TRUE, places all strings in ROM.
Switch_To_Rom	FALSE	If TRUE, places all switches in ROM.

Table continues on the next page...

Table 4-2. Application File Options (continued)

Option	Default Value	Description
Uninit_Globals_To_Bss	FALSE	If TRUE, places all uninitialized global variables to BSS.
Uninit_And_Zeroed_Globals_To_Bss	FALSE	If TRUE, places all uninitialized or zero initialized global variables to BSS.

4.1.3 Pre-processing control options

Table 4-3 lists the shell control pre-processing options.

Table 4-3. Shell control pre-processing options

Option	Description
`-C [file]`	Pre-processes only; preserves comments
`-Dmacro`	Defines specified pre-processor macro (No space required before the macro name)
`-Idir`	Adds the specified directory to the `include` file paths (No space required before the directory name)
`-M [file]`	Pre-processes only; generates dependencies in the `make` syntax
`-Umacro`	Un-defines specified pre-processor macro (No space required before the macro name)

4.1.4 Output file extension options

Table 4-4 lists the shell control output file options.

Table 4-4. Shell control output file options

Option	Description
`-xasm file`	Treats the specified file as `.asm` assembler source file
`-xc file`	Treats the specified file as `C` source file
`-o` *file*	Assigns the specified name to the output file
`-r` *dir*	Redirects all output to the specified directory

4.1.5 C language options

Table 4-5 lists the C language options.

Table 4-5. C Language Options

Option	Description
-ansi	Specifies strict ANSI mode; all C source files must be ANSI/ISO C files with no extensions(Default mode is ANSI/ISO C with extensions)
-g	Adds DWARF debug information to the generated files
-ge	Adds DWARF debug extensions to the generated files
-kr	Specifies K&R/pcc mode; all C source files must be K&R/pcc C files (Default mode is ANSI/ISO C with extensions)
--min_enum_size<n>	Sets minimum enumeration-type size to the specified number of bytes: 1, 2, or 4 (4 is the default option)
--min_struct_align=<min>	Sets minimum alignment for structures. Default alignment value is 1
-reject_floats	Checks for floating-point variables/operations. If found, stops compilation and issues an error message.
-requireprotos	Controls whether or not the compiler should expect function prototypes
-sc	Makes char type variables signed (Default mode)
-usc	Makes char type variables unsigned

4.1.6 Optimization and code options

Table 4-6 lists the optimization and code options.

Table 4-6. Optimization and code options

Option	Description
-align	Forces alignment:0 = disables alignment1= aligns hardware loops2 = aligns hardware and software loops3 = aligns all labels4 = aligns all labels and function call return points
-allow_load_spec	Enables memory read speculation
-opt=[no]alias_by_type	Disables alias by type rules when analyzing aliases.
-mod	Enables modulo buffer support
-no_load_spec	Disables memory read speculation
--no_switch_table	Blocks generation of a switch table

Table continues on the next page...

Table 4-6. Optimization and code options (continued)

Option	Description
`-O (or -O2)`	Performs all optimizations with high-level local register allocation; produces optimized, non-linear assembly language code (Default optimization level)
`-O0`	Disables all optimizations; produces unoptimized assembly language code
`-O1`	Performs all target-independent optimizations but no target-specific optimizations
`-O3`	Performs enhanced register allocation
`-O4`	Performs both target-independent and target-dependent optimizations.
`-Od`	Disables all optimizations; produces smaller code size than `-O0` option
`-Og`	Performs optimization across source file boundaries
`-Os`	Adds space optimization to the existing level of optimization; reduces size of assembly code (Not valid with `-O0` option); when used in conjunction with `-O3`, generates the smallest code

4.1.7 Shell passthrough options

Table 4-7 lists the shell passthrough options.

Table 4-7. Shell passthrough options

Option	Description
`-Xasm option`	Passes specified options and arguments to the assembler
`-Xll t option`	Passes specified options and arguments to the low-level optimizer.
`-Xlnk option`	Passes specified options and arguments to the linker

4.1.8 File and message output options

Table 4-8 lists the file and message output options.

Table 4-8. File and message output options

Option	Description
-de	Keeps generated error file for each source file
-dL [file]	Generates default C list file for each (or specified) source file
-dL1 [file]	Generates a C list file with include files for each (or specified) source file. Note that the list file will not be generated for the include file enclosed between angle brackets, <>.
-dL2 [file]	Generates a C list file with expansions for each (or specified) source file
-dL3 [file]	Generates a C list file with include files and expansions for each (or specified) source file
-dm [file]	Generates a link map file
-do	Generates structure field definition offsets as EQUs
-dx [file]	Generates a cross-reference information file
-n	Displays command lines without executing
-q or -w	Displays errors only; quite mode (Default setting)
-s	Instructs the compiler to keep all the assembly language (.sl) files that it generates
-v	Displays all information; verbose mode
-Wparameter-<keyword> or-W<index>	Controls warnings and remarks
-save-temps	Continues the build process without deleting the intermediate files (keeps both assembly and preprocessed files).

4.1.9 Hardware configuration options

Table 4-9 lists the hardware configuration options.

Table 4-9. Hardware Configuration Options

Option	Description
-arch	Specifies target architecture; valid options are: (cores) sc3900fp, b4460, b4860. Note that Freescale silicon is big-endian.
allconst_to_rom	Stores in ROM all symbols that the high-level optimizer finds constant. Default value is off.
init_to_rom	Permits placement of an array initializer in ROM. Default value is off.
-be	Compiles for big-endian target configuration
-crt file	Specifies user-defined startup file

Table continues on the next page...

Table 4-9. Hardware Configuration Options (continued)

Option	Description
`-ma file`	Specifies user-defined application configuration file
`-mb`	Compiles in big memory model
`-mb1`	Compiles in big memory model with runtime library calls in huge memory model
`-mem file`	Specifies user-defined linker command file
`-mrom`	Copies all the initialized variables from ROM to RAM at startup
`-view`	Selects a specific view for the application configuration file

Related Tasks

- How to enable support for SC3900FP compiler
- How to disable hardware floating point support in SC3900FP compiler
- How to enable fused multiply and add generation

4.1.10 Library options

Table 4-10 lists the library options.

Table 4-10. Library Options

Option	Description
`-complib [`*name*`]`	Creates a self-contained component with the specified name
`const_to_rom`	Places constants in ROM. Default value is on.
`-l` *file*	Supplies the specified library file
`-no_runtime_lib`	Does not link with default runtime libraries
`-npr`	Disables the replacement of prototype (intrinsic) function
`-reentrant`	Links with reentrant library and the startup file
`-selflib`	Creates a self-contained library
`string_to_rom`	Places all strings in ROM. Default value is off.
`switch_to_rom`	Places all switches in ROM. Default value is off.
`uninit_globals_to_bss`	Places all uninitialized global variables to BSS. Default value is off.
`uninit_and_zeroed_globals_to_bss`	Places all uninitialized or zero initialized global variables to BSS. Default value is off.

4.1.11 Low level optimizer options

Low level optimizer (`-Xllt`) options are divided into two categories:

- Basic options
- Advanced options

Table 4-11 lists the low level optimizer basic options.

Table 4-11. Low level optimizer basic options

Option	Description
`-static_asm`	Implicitly assume that assembly functions are static
`-do`	Generates structure field definitions offset as EQUs

4.1.12 Compiler front-end warning messages

The table below lists the keywords that you use for compiler front-end warning reporting.

Table 4-12. Compiler front-end warning reporting keywords

Keyword	Command-line option	Default value	Description
`on`	`-Wmwfe-on`		Enables most compiler front-end warnings
`off`	`-Wmwfe-off`		Disables all compiler front-end warnings
`all`	`-Wmwfe-all`		Enables all compiler front-end warnings
`most`	`-Wmwfe-most`		Enables most compiler front-end warnings
`full`	`-Wmwfe-full`		Enables all compiler front-end warnings, but may lead to spurious warnings
`anyptrintconv`	`-Wmwfe-anyptrintconv`	False	Generates warnings about any pointer-to-integer conversions.
	`-Wnomwfe-anyptrintconv`		Does not generate warnings about any pointer-to-integer conversions.
`cmdline`	`-Wmwfe-cmdline`	False	Enables command-line driver/parser warnings.
	`-Wnomwfe-cmdline`		Disables command-line driver/parser warnings.
`comma`	`-Wmwfe-comma`	False	Generates warnings about extra commas.
	`-Wnomwfe-comma`		Does not generate warnings about extra commas.
`display`	`-Wmwfe-display`	False	Displays list of active warnings.
`dump`	`-Wmwfe-dump`	False	Displays list of active warnings.

Table continues on the next page...

Table 4-12. Compiler front-end warning reporting keywords (continued)

Keyword	Command-line option	Default value	Description
emptydecl	-Wmwfe-emptydecl (or -Wmwfe-empty)	True	Generates warnings about empty declarations.
	-Wnomwfe-emptydecl (or -Wnomwfe-empty)		Does not generate warnings about empty declarations.
error	-Wmwfe-error (or -Wmwfe-err)		Activates treatment of compiler-front-end warnings as errors.
	-Wnomwfe-error (or -Wnomwfe-err)		Deactivates treatment of compiler-front-end warnings as errors.
extended	-Wmwfe-extended	True	Activates pedantic error checking.
	-Wnomwfe-extended		Deactivates pedantic error checking
extracomma	-Wmwfe-extracomma	False	Generates warnings about extra commas.
	-Wnomwfe-extracomma		Does not generate warnings about extra commas
filecaps	-Wmwfe-filecaps	False	Generates warnings about incorrect capitalization in #include "".
	-Wnomwfe-filecaps		Does not generate warnings about incorrect capitalization in #include "".
gen_asm_header_error	-Wmwfe-no-gen_asm_header_error	True	Transforms error for "inline assembly functions.
	-Wnomwfe-gen_asm_header_error		Does not support 'asm_header' section" into warning
hidevirtual	-Wmwfe-hidevirtual	True	Generates warnings about hidden virtual functions.
	-Wnomwfe-hidevirtual		Does not generate warnings about hidden virtual functions.
hiddenvirtual	-Wmwfe-hiddenvirtual (or -Wmwfe-hidden)	True	Generates warnings about hidden virtual functions.
	-Wnomwfe-hidevirtual (or _Wmwfe-hidden)		Does not generate warnings about hidden virtual functions.
illpragmas	-Wmwfe-illpragmas	True	Generates warnings about invalid pragmas.
	-Wnomwfe-illpragmas		Does not generate warnings about invalid pragmas.
implicitconv	-Wmwfe-implicitconv (or -Wmwfe-implicit)	False	Generates warnings about implicit integer-to-floating-point or floating-point-to-integer conversions.
	-Wnomwfe-implicitconv (or -Wnomwfe-implicit)		Does not generate warnings about implicit integer-to-floating-point or floating-point-to-integer conversions.
impl_float2int	-Wmwfe-impl_float2int	False	Generates warnings about implicit floating-point-to-integer conversions.
	-Wnomwfe-impl_float2int		Does not generate warnings about implicit floating-point-to-integer conversions.
impl_int2float	-Wmwfe-impl_int2float	Flase	Generates warnings about implicit integer-to-floating-point conversions.
	-Wnomwfe-impl_int2float		Does not generate warnings about implicit integer-to-floating-point conversions.

Table continues on the next page...

Table 4-12. Compiler front-end warning reporting keywords (continued)

Keyword	Command-line option	Default value	Description
impl_signedunsigned	-Wmwfe-impl_signedunsigned	False	Generates warnings about implicit signed/unsigned conversions.
	-Wnomwfe-Impl_signedunsigned		Does not generate warnings about implicit signed/unsigned conversions.
iserror	-Wmwfe-iserror (or -Wmwfe-iserr)	Off	Activates treatment of compiler-front-end warnings as errors.
	-Wnomwfe-iserror (or -Wnomwfe-iserr)		Deactivates treatment of compiler-front-end warnings as errors.
largeargs	-Wmwfe-largeargs	False	Generates warnings about passing large arguments to un-prototyped functions.
	-Wnomwfe-largeargs		Does not generate warnings about passing large arguments to un-prototyped functions.
missingreturn	-Wmwfe-missingreturn	True	Generates warnings about returns without values in non-void-returning functions.
	-Wnomwfe-missingreturn		Does not generate warnings about returns without values in non-void-returning functions.
notinlined	-Wmwfe-notinlined	False	Generates warnings about `inline' functions not inlined.
	-Wnomwfe-notinlined		Does not generate warnings about `inline' functions not inlined.
notused	-Wmwfe-notused	False	Generates warnings about result of non-void-returning function not being used.
	-Wnomwfe-notused		Does not generate warnings about result of no-void-returning function not being used.
padding	-Wmwfe-padding	False	Generates warnings about padding added between struct members.
	-Wnomwfe-padding		Does not generate warnings about padding added between struct members.
pedantic	-Wmwfe-pedantic	True	Activates pedantic error checking
	-Wnomwfe-pedantic		Deactivates pedantic error checking
possible	-Wmwfe-possible	True	Generates warnings about possible unwanted side effects.
	-Wnomwfe-possible		Does not generate warnings about possible unwanted side effects.
pragmas	-Wmwfe-pragmas	True	Generates warnings about invalid pragmas
	-Wnomwfe-pragmas		Does not generate warnings about invalid pragmas
ptrintconv	-Wmwfe-ptrintconv	True	Generates warnings about lossy pointer-to-integer conversions.
	-Wnomwfe-ptrintconv		Does not generate warnings about lossy pointer-to-integer conversions.
structclass	-Wmwfe-structclass	False	Generates warnings about inconsistent use of class and struct.
	-Wnomwfe-structclass		Does not generate warnings about inconsistent use of class and struct.

Table continues on the next page...

Table 4-12. Compiler front-end warning reporting keywords (continued)

Keyword	Command-line option	Default value	Description
sysfilecaps	-Wmwfe-sysfilecaps	False	Generates warnings about incorrect capitalization in #include <>.
	-Wnomwfe-sysfilecaps		Does not generate warnings about incorrect capitalization in #include <>.
tokenpasting	-Wmwfe-tokenpasting	True	Generates warnings about tokens not formed by the ## operator.
	-Wnomwfe-tokenpasting		Does not generate warnings about tokens not formed by the ## operator.
undefmacro	-Wmwfe-undefmacro (or -Wmwfe-undef)	False	Generates warnings about undefined macros in #if/#else conditionals.
	-Wnomwfe-undefmacro (or -Wnomwfe-undef)		Does not generate warnings about undefined macros in #if/#else conditionals.
unwanted	-Wmwfe-possible	True	Generates warnings about possible unwanted side effects
	-Wnomwfe-possible		Does not generate warnings about possible unwanted side effects
unused	-Wmwfe-unused	True	Generates warnings about unused arguments or variables.
	-Wnomwfe-unused		Does not generate warnings about unused arguments or variables.
unusedarg	-Wmwfe-unusedarg	False	Generates warnings about unused arguments
	-Wnomwfe-unusedarg		Does not generate warnings about unused arguments.
unusedexpr	-Wmwfe-unusedexpr	False	Generates warnings about using expressions as statements without side effects.
	-Wnomwfe-unusedexpr		Does not generate warnings about using expressions as statements without side effects.
unusedvar	-Wmwfe-unusedvar	True	Generates warnings about unused variables.
	-Wnomwfe-unusedvar		Does not generate warnings about unused variables.

NOTE

For complete description and possible solutions of warning messages generated by these warning keywords, refer to the *Build Tools Message Reference Manual*.

4.1.13 Warning index values

Table 4-13 lists the warning index values that you can use with `-W` option.

Table 4-13. Warning Index Values

Index Value	Command-line Option	Description
1401	`-W1401` `-Wno1401`	Generates wrapper warning: "A tool that is switched off has an option"Does not generate this wrapper warning.
1402	`-W1402` `-Wno1402`	Generates wrapper warning: "A tool has finished abnormally"Does not generate this wrapper warning.
1403	`-W1403` `-Wno1403`	Generates wrapper warning: "A file is defined as being parsed by mwfe and forced to cobj"Does not generate this wrapper warning.
1404	`-W1404` `-Wno1404`	Generates wrapper warning: "A file is defined as being parsed by cobj and forced to mwfe"Does not generate this wrapper warning.
1405	`-W1405` `-Wno1405`	Generates wrapper warning: "Lister Generation failed"Does not generate this wrapper warning.
1407	`-W1407` `-Wno1407`	Generates wrapper warning: "Misuse of -env option"Does not generate this wrapper warning.
1408	`-W1408` `-Wno1408`	Generates wrapper warning: "An option appears twice"Does not generate this wrapper warning.
1409	`-W1409` `-Wno1409`	Generates wrapper warning: "An option is defined in a duplicated way"Does not generate this wrapper warning.
1410	`-W1410` `-Wno1410`	Generates wrapper warning: "The command line calls nothing"Does not generate this wrapper warning.
1411	`-W1411` `-Wno1411`	Generates wrapper warning: "An input file will not be treated"Does not generate this wrapper warning.
1412	`-W1412` `-Wno1412`	Generates wrapper warning: "An unknown extension for file is used, so file is treated as an input to the linker"Does not generate this wrapper warning.
1413	`-W1413` `-Wno1413`	Generates wrapper warning: "Internal Warning Message"Does not generate this wrapper warning.
1601	`-W1601` `-Wno1401`	Generates wrapper warning: "A tool that is switched off has an option."Does not generate this wrapper warning.
1401	`-W1401` `-Wno1601`	Generates wrapper remark: "Debugging in optimized mode."Does not generate this wrapper remark.

Table continues on the next page...

Table 4-13. Warning Index Values (continued)

Index Value	Command-line Option	Description
1602	-W1602 -Wno1602	Generates wrapper remark: "Default name for the error file is used"Does not generate this wrapper remark.
1603	-W1603 -Wno1603	Generates wrapper remark: "Default name for the list file is used"Does not generate this wrapper remark.
1604	-W1604 -Wno1604	Generates wrapper remark: "Default name for the crossref file is used"Does not generate this wrapper remark.
3206	-W3206 -Wno3206	Generates Icode warnings while performing standard optimizations.Does not generate Icode warnings while performing standard optimizations.
3207	-W3207 -Wno3207	Generates Icode warnings while performing loop optimizations.Does not generate Icode warnings while performing loop optimizations.
3208	-W3208 -Wno3208	Generates Icode warnings while performing range analysis optimizations.Does not generate Icode warnings while performing range analysis optimizations.
3209	-W3209 -Wno3209	Generates Icode warnings while performing code generation optimizations.Does not generate Icode warnings while performing code generation optimizations.
3210	-W3210 -Wno3210	Generates Icode warnings upon reaching maximum stack size.Does not generate Icode warnings upon reaching maximum stack size.
3211	-W3211 -Wno3211	Generates Icode warnings upon exhausting memory.Does not generate Icode warnings upon exhausting memory.
3001	-W3001 -Wno3001	Generates Icode default remarks.Does not generate Icode default remarks.
3002	-W3002 -Wno3002	Generates Icode command-line option remarks.Does not generate Icode command-line option remarks.
3003	-W3003 -Wno3003	Generates internal Icode option remarks.Does not generate internal Icode option remarks.
3004	-W3004 -Wno3004	Generates Icode segment remarks.Does not generate Icode segment remarks.
3005	-W3005 -Wno3005	Generates Icode remarks while parsing.Does not generate Icode remarks while parsing.

Table continues on the next page...

Table 4-13. Warning Index Values (continued)

Index Value	Command-line Option	Description
3006	-W3006 -Wno3006	Generates Icode remarks while performing standard optimizations.Does not generate Icode remarks while performing standard optimizations.
3007	-W3007 -Wno3007	Generates Icode remarks while performing loop optimizations.Does not generate Icode remarks while performing loop optimizations.
3008	-W3008 -Wno3008	Generates Icode remarks while performing range analysis optimizations.Does not generate Icode remarks while performing range analysis optimizations.
3009	-W3009 -Wno3009	Generates Icode remarks while performing code generation optimizations.Does not generate Icode remarks while performing code generation optimizations.
3010	-W3010 -Wno3010	Generates Icode remarks upon reaching maximum stack size.Does not generate Icode remarks upon reaching maximum stack size.
3011	-W3011 -Wno3011	Generates Icode remarks upon exhausting memory.Does not generate Icode remarks upon exhausting memory
6001	-W6001 -Wno6001	Generates linker command-line warnings.Does not generate linker command-line warnings.
6002	-W6002 -Wno6002	Generates linker warnings while performing dead-stripping.Does not generate linker warnings while performing dead-stripping.
6003	-W6003 -Wno6003	Generates linker warnings while processing the linker command file.Does not generate linker warnings while processing the linker command file.
6999	-W6999 -Wno6999	Generates default linker warnings.Does not generate default linker warnings.

4.2 Pragmas and attributes

In this appendix:

- Function pragmas and attributes

- Statement pragmas and attributes
- Other pragmas and attributes

4.2.1 Function pragmas and attributes

This section lists pragmas and attributes that apply to specific functions. You must define these pragmas in the scope of their functions, directly after the open-brace character ({) that marks the start of the scope.

The attributes are defined before the open-brace character ({).

4.2.1.1 #pragma alias_by_type

Instructs the compiler to use alias by type rules for alias analysis.

Syntax

```
#pragma alias_by_type on | off | reset
```

Remarks

Refer Chapter 6.5, Paragraph 7 in C99 Standard document, to understand alias by type rules. By default, this pragma is on.

4.2.1.2 #pragma fct_never_return func_name

Deprecated. The compiler will ignore the pragma and issue a warning. Use the `__attribute__((noreturn))` keyword.

Permits non-standard return from the specified function; enables optimization that involves removal of code that includes normal return instructions. You may place this pragma in a header file, or at the head of a source file.

4.2.1.3 #pragma inline

Deprecated. The compiler will ignore the pragma and issue a warning. Use the `inline/__inline` keyword.

Directs the compiler to inline any function that contains this pragma. Function inlining must not to be confused with `#pragma inline`. Former is compiler implicit and latter is user enforced. The user can enforce function-inlining using `#pragma inline` and several other methods and such enforced inlining works at `-O0` as well.

However, the compiler implicit function-inlining is triggered only on `-O2` and higher.

4.2.1.4 #pragma inline_call func_name

Directs the compiler to inline the next call of the specified function; must be placed just before that call. (Has no effect on a function call made through a pointer.)

4.2.1.5 #pragma interrupt func_name

Defines the specified function as an interrupt handler; makes the function an interrupt entry.

4.2.1.6 #pragma never_return

Deprecated. The compiler will ignore the pragma and issue a warning. Use the `__attribute__((noreturn))` keyword.

Permits non-standard return from the function that contains this pragma; enables optimization that involves removal of code that includes normal return instructions.

4.2.1.7 #pragma noinline

Deprecated. The compiler will ignore the pragma and issue a warning. Use the `__attribute__((noinline))` keyword.

Directs the compiler to *not* inline any function that contains this pragma.

4.2.1.8 #pragma novector and __attribute__((novector))

Disables compiler automatic vectorization at loop level (`#pragma novector`) or function level (`__attribute__((novector))`).

Syntax

```
#pragma novector
```

Example

```
for (i=0; i<100; i++)
{
#pragma novector
dst[i] = src[i];
}
```

If the pragma is added, the loop code is not vectorized even if the compiler considers it is legal to vectorize the code. If the pragma is removed, the loop code is vectorized if the compiler considers it is legal cu vectorize the code. This fully disables all vectorization optimizations on the loop code.

If `__attribute__((novector))` is added to the function description, it fully disables vectorization for the considered function. It is not propagated after function inline, so it must be always associated to `__attribute__((noinline))`.

Syntax

```
__attribute__((novector))
```

Example

```
void func(int arg) __attribute__((novector)) __attribute__((noinline))
{
                ...
}
```

Related Task

- How to disable automatic vectorization

4.2.1.9 Pragmas to control function inlining

Following pragmas control the function inlining:

- #pragma auto_inline
- #pragma dont_inline

4.2.1.9.1 #pragma auto_inline

Instructs the compiler to automatically select the functions suitable for inlining, in addition to the functions declared explicitly with the inline keyword.

Syntax

```
#pragma auto_inline on | off | reset
```

Remarks

Note that if you enable the `dont_inline` pragma, the compiler ignores the setting of the `auto_inline` pragma and does not inline any functions.

By default, this pragma is `on` starting with O2.

4.2.1.9.2 #pragma dont_inline

Instructs the compiler to not to inline any function calls, even those declared with the `inline` keyword or within a class declaration. Also, the compiler does not automatically inline functions, regardless of the `auto_inline` pragma.

Syntax

```
#pragma dont_inline on | off | reset
```

Remarks

If you disable this pragma, the compiler expands all inline function calls, within the limits you set through other inlining-related pragmas.

By default, this pragma is off.

4.2.1.9.3 #pragma inline_max_auto_size

Determines the maximum complexity for an auto-inlined function.

Syntax

```
#pragma inline_max_auto_size (complex)
```

Parameters

`complex`

The `complex` value multiplied by 5 is an approximation of the number of statements in a function. The current default value is 300 (60 for `inline_max_auto_size`). Selecting a higher value will inline more functions, but can lead to excessive code bloat.

Remarks

This pragma does not correspond to any panel setting.

4.2.1.9.4 #pragma ipa_inline_max_auto_size

Determines the maximum complexity for an auto-inlined function if called only once.

Syntax

```
#pragma ipa_inline_max_auto_size (complex)
```

Parameters

`complex`

The `complex` value is an approximation of the number of statements in a function. The current default value is 650. Selecting a higher value will inline more functions, but can lead to excessive code bloat.

Remarks

This pragma does not correspond to any panel setting.

4.2.1.9.5 #pragma warn_notinlined

Controls the issuing of warning messages for functions the compiler cannot inline.

Syntax

```
#pragma warn_notinlined on | off | reset
```

Remarks

The compiler issues a warning message for non-inlined inline (i.e., on those indicated by the `inline` keyword or in line in a class declaration) function calls.

By default, this pragma is off.

4.2.1.9.6 #pragma aggressive_inline

Instructs the IPA based inliner to inline more functions when this option is enabled.

Syntax

```
#pragma aggressive_inline on | off | reset
```

Remarks

Enabling this option can cause code bloat in programs that overuse inline functions.

By default, this pragma is off.

4.2.2 Statement pragmas and attributes

You place pragmas and attributes that apply to statements immediately before their relevant statements.

4.2.2.1 #pragma align func_name al_val

Deprecated. The compiler will ignore the pragma and issue a warning.

Use the `__attribute__((aligned(<n>)))` keyword.

Applies specified alignment value to specified function; overrides compiler's default alignment rule. Also applies to alignment given through an application file.

The al_val must be a power of 2, limit is 8192 (error emitted by Farscape otherwise: "Error: illegal or unsupported alignment value").

4.2.2.2 #pragma noswitchtable

Forces mapping of a C switch statement, using a set of conditional static gotos, instead of a computed goto.

Remarks

Must be placed just before the switch statement.

4.2.2.3 #pragma profile value

Sets profiling information for a statement.

Remarks

`value` is a signed 32-bit value. Its maximum is therefore `0x7FFFFFFF`.

4.2.2.4 #pragma relax_restrict

Controls the use of `restrict` qualifier on nested variables.

The optimizer ignores "" qualifiers on nested variables, because (unlike parameters) they can alias objects in the same function that are defined outside of the scope of the restrict variable or even within the same scope but in another iteration (if the nested variable is defined in a loop).

Remarks

Enabling `relax_restrict` instructs the compiler to treat local "restrict" pointers as pointers that will not alias any object that is used within the scope of the containing function. So this may generate incorrect code for cases where nested restrict variables are used to access objects from outer scopes or across iterations.

By default, `relax_restrict` is disabled.

4.2.2.5 #pragma require_prototypes on/off

Enforces the requirement of function prototypes. The compiler generates an error message if you define a previously referenced function that does not have a prototype. If you define the function before it is referenced but do not give it a prototype, this pragma causes the compiler to issue a warning message.

4.2.2.6 #pragma switchtable

Forces mapping of a C switch statement, using a computed jump; uses pointer type as default. Must be placed just before the switch statement. Note that the compiler throws an error message for `#pragma switchtablebyte` if the switch-statement > 256 bytes.

4.2.2.7 #pragma switchtablebyte

Like `#pragma switchtable`, provided that code generated for lines between the switch-statement braces ({ }) fits into 255 bytes. Must be placed just before the switch statement. Note that the compiler throws an error message if the switch-statement > 256 bytes.

4.2.2.8 #pragma switchtableword

Like `#pragma switchtable`, provided that code generated for lines between the switch-statement braces ({ }) fits into 65535 bytes. Must be placed just before the switch statement.

4.2.2.9 #pragma no_btb

Forces the compiler to not to `jump` resulting from an `IF` statement in the branch target buffer. Therefore, no branch prediction occurs on resulting `jump` or branch instruction. Useful to avoid penalties that result from wrong prediction if the condition in the `IF` statement is random in nature.

4.2.3 Other pragmas and attributes

Place these additional pragmas and attributes immediately after the definition of the objects to which the pragmas refer. The object definitions must be explicit.

4.2.3.1 #pragma align var_name al_val | *ptr al_val

Deprecated. The compiler will ignore the pragma and issue a warning. Use the `__attribute__((aligned(<n>)))` keyword.

Forces alignment on an array or pointer variable object.

For local variables, al_val must be a power of 2, limit is 8192 (error emitted by Farscape otherwise: "Error: illegal or unsupported alignment value").

For global variables, al_val must be a power of 2, limit is 8192 (error emitted by Farscape otherwise: "Error: illegal or unsupported alignment value").

4.2.3.2 #pragma bss_seg_name "name"

Renames bss segment in the ELF file. (You must define the name used to override the default in the linker command file.)

4.2.3.3 #pragma data_seg_name "name"

Renames data segment in the ELF file. (You must define the name used to override the default in the linker command file.)

4.2.3.4 #pragma init_seg_name "name"

Renames init segment in the ELF file. (You must define the name used to override the default in the linker command file.)

4.2.3.5 #pragma min_struct_align min

Sets minimum alignment for structures; applies to the file in which it is defined. Default alignment value is 4. However, note that using a value lower than the default value of 4 may break the Application Binary Interface (ABI).

4.2.3.6 #pragma loop_count (min_iter, max_iter [, {modulo}, [remainder]])

Specifies:

- Minimum number of iterations (the compiler can use this value to remove loop bypass tests),
- Maximum number of iterations (the compiler can use this value to assess the induction-variable range),
- Modulo/remainder pair (the compiler uses these values to unroll loops with dynamic loop count).

Example

Iterate the loop at least once, at most 16 times; the loop count is a multiple of 2:

```
#pragma loop_count (1,16,2,0)
```

Remarks

This pragma must be inserted after the loop starter; not all fields are mandatory.

4.2.3.7 #pragma loop_multi_sample constant_val

For an enclosed loop nest, performs an unroll and jam, then unrolls the enclosed loop. The constant is the unroll-and-jam factor. This #pragma applies only to loop nests that are single instruction blocks without calls.

Example:

```
#pragma loop_multi_sample 8
```

4.2.3.8 #pragma loop_unroll constant_val

Unrolls constant time in the loop in which this pragma appears. Applies only to a single instruction block loop without calls.

Example:

```
#pragma loop_unroll 2
```

4.2.3.9 #pragma loop_unroll_and_jam constant_val

Performs an unroll and jam on the enclosed loop nest. The constant is the unroll-and-jam factor. This #pragma applies only to loop nests that are single instruction blocks, without calls.

Example:

```
#pragma loop_unroll_and_jam 8
```

4.2.3.10 #pragma pgm_seg_name "name"{, "overlay"}

Renames text segment in the ELF file. (You must define the name used to override the default in the linker command file.)

4.2.3.11 #pragma rom_seg_name "name"{, "overlay"}

Renames the `rom` segment in the ELF file. (You must define the name used to override the default in the linker command file.)

4.3 Runtime libraries

This appendix describes runtime libraries available with the compiler.

- Character typing and conversion (ctype.h)
- Floating-point characteristics (float.h)
- Floating-Point library interface (fltmath.h)
- Integer characteristics (limits.h)
- Locales (locale.h)
- Floating-point math (math.h)
- Program administrative functions
- I/O library (stdio.h)
- General utilities (stdlib.h)
- String functions (string.h)
- Time functions (time.h)

Table 4-14 summarizes the C libraries. The non-ISO C libraries contain the compiler's built-in intrinsic functions. The header file you use to include such a library depends on whether your code has any conflicts between certain assembly-language operations and intrinsic functions.

Table 4-14. Supported Libraries

Library Type	Header File	Description
ISO	`ctype.h`	Character typing and conversion
	`float.h`	Floating-point characteristics
	`getopt.h`	`getopt()` function implementation
	`limits.h`	Integer characteristics
	`locale.h`	Locales
	`math.h`	Floating-point math
	`setjmp.h`	Nonlocal jumps
	`signal.h`	Signal handling
	`stdarg.h`	Variable arguments
	`stddef.h`	Standard definitions
	`stdio.h`	I/O library
	`stdlib.h`	General utilities
	`string.h`	String functions
	`time.h`	Time functions
Non-ISO	`prototype.h`	Built-in intrinsic functions.
	`prototype_asm.h`	Alternative header file that defines built-in intrinsic functions so that they do not conflict with these assembler operations: `add`, `debug`, `debugv`, `di`, `ei`, `max`, `mark`, `min`, `mpyuu`, `mpysu`, `mpyus`, `stop`, `sub`, `trap`, `wait`. If needed, include this file in your code instead of `prototype.h`.

4.3.1 Character typing and conversion (ctype.h)

Table 4-15 lists the testing and conversion functions that the compiler supports. These functions are in the `ctype.h` library.

Table 4-15. Testing and conversion functions

Category	Function	Purpose
Testing functions	`int isalnum(int)`	Tests for `isalpha` or `isdigit`
	`int isalpha(int)`	Tests for `isupper` or `islower`

Table continues on the next page...

Table 4-15. Testing and conversion functions (continued)

Category	Function	Purpose
	`int iscntrl(int)`	Tests for any control character
	`int isdigit(int)`	Tests for decimal digit character
	`int isgraph(int)`	Tests for any printing character except space
	`int islower(int)`	Tests for lowercase alphabetic character
	`int isprint(int)`	Tests for any printing character including space
	`int ispunct(int)`	Tests for any printing character not space and not `isalnum`
	`int isspace(int)`	Tests for white-space characters
	`int isupper(int)`	Tests for uppercase alphabetic character
	`int isxdigit(int)`	Tests for hexadecimal digit character
Conversion functions	`int tolower(int)`	Converts uppercase alphabetic character to the equivalent lower case character
	`int toupper(int)`	Converts lowercase alphabetic character to the equivalent uppercase character

4.3.2 Floating-point characteristics (float.h)

The compiler uses IEEE format (ANSI/IEEE Std 754-1985) to represent single precision and double precision floating point numbers. The double precision floating-point format is supported by default.

Table 4-16 lists the name and default value of each single precision floating point manifest constant.

NOTE

The floating-point constant definitions are in `float.h`.

Table 4-16. Default values of single precision floating-point constants

Constant	Value	Purpose
`FLT_DIGDBL_DIGLDBL_DIG`	`666`	Number of decimal digits of precision
`FLT_EPSILONDBL_EPSILONLDBL_EPSILON`	`1.1920929E-071.1920929E-071.1920929E-07`	Minimum positive number c such that 1.0 + c does not equal 1.0
`FLT_MANT_DIGDBL_MANT_DIGLDBL_MANT_DIG`	`242424`	Number of base-2 digits in the mantissa
`FLT_MAX_10_EXPDBL_MAX_10_EXPLDBL_MAX_10_EXP`	`383838`	Maximum positive integers *n* such that 10n is representable

Table continues on the next page...

Table 4-16. Default values of single precision floating-point constants (continued)

Constant	Value	Purpose
FLT_MAX_EXPDBL_MAX_EXPLDBL_MAX_EXP	128128128	Maximum positive integer *n* such that 2n-1 is representable
FLT_MAXDBL_MAXLDBL_MAX	3.4028235E+383.4028235E+383.4028235E+38	Maximum positive floating-point number
FLT_MIN_10_EXPDBL_MIN_10_EXPLDBL_MIN_10_EXP	-39-39-39	Minimum negative integer *n* such that 10n is representable
FLT_MIN_EXPDBL_MIN_EXPLDBL_MIN_EXP	-126-126-126	Minimum negative integer *n* such that 2n-1 is representable
FLT_MINDBL_MINLDBL_MIN	5.8774817E-395.8774817E-395.8774817E-39	Minimum positive number
FLT_RADIXFLT_ROUNDS	21	Floating-point exponent is expressed n radix 2. Floating-point rounding is to nearest even number.

Table 4-17 lists the values of these constants for double precision floating points.

Table 4-17. Default values of double precision floating-point constants

Constant	Value	Purpose
FLT_DIGDBL_DIGLDBL_DIG	61010	Number of decimal digits of precision
FLT_EPSILONDBL_EPSILONLDBL_EPSILON	1.1920929E-072.220446049250313 1E-162.2204460492503131e-16	Minimum positive number c such that 1.0 + c does not equal 1.0
FLT_MANT_DIGDBL_MANT_DIGLDBL_MANT_DIG	245353	Number of base-2 digits in the mantissa
FLT_MAX_10_EXPDBL_MAX_10_EXPLDBL_MAX_10_EXP	38308308	Maximum positive integers *n* such that 10n is representable
FLT_MAX_EXPDBL_MAX_EXPLDBL_MAX_EXP	12810241024	Maximum positive integer *n* such that 2n-1 is representable
FLT_MAXDBL_MAXLDBL_MAX	3.4028235E+381.79769313e+3081.79769313e+308	Maximum positive floating-point number
FLT_MIN_10_EXPDBL_MIN_10_EXPLDBL_MIN_10_EXP	-39-308-308	Minimum negative integer *n* such that 10n is representable
FLT_MIN_EXPDBL_MIN_EXPLDBL_MIN_EXP	-126-1022-1022	Minimum negative integer *n* such that 2n-1 is representable
FLT_MINDBL_MINLDBL_MIN	5.8774817E-394.94065645e-3084.94065645e-308	Minimum positive number
FLT_RADIXFLT_ROUNDS	21	Floating-point exponent is expressed n radix 2. Floating-point rounding is to nearest even number.

4.3.3 Floating-Point library interface (fltmath.h)

Header file `fltmath.h` defines the software floating-point library interface. The code generator (of the compiler for floating-point expression evaluation) calls most of these functions. User code also may call these functions directly.

The floating-point library supports the IEEE-754 single-precision floating-point standard, with the following exceptions:

- Only round-to-nearest-even rounding method, IEEE default rounding, is supported in both software and hardware floating point libraries.
- The hardware floating point library has two additional differences:
  - De-normalized input is considered by SC3900FP to have zero value, that is single precision numbers are flushed to zero. No de-normalized numbers other than zero are generated as a calculation result.
  - No hardware exceptions are automatically generated.

4.3.3.1 Round_Mode

The compiler supports following rounding mode:

- `ROUND_TO NEAREST_EVEN` - (Default) The representable value nearest to the infinitely precise intermediate value is the result. If the two nearest representable values are equally near (tie), then the one with the least significant bit equal to zero (even) is the result.

4.3.3.2 IEEE_Exceptions

NOTE

IEEE Exceptions are valid only for the software floating-point implementation.

NOTE

The hardware floating point library does not support flags and automatic generation of the hardware exceptions.

For the hardware floating point library, the floating point related flags are located in the GCR register. The following flags are supported: Float Inexact Flag (FINEX), Float Invalid Flag (Finval), Float Division by Zero Flag (FDIVZ), Float Overflow Flag (FOVER), Float Underflow Flag (FUNDR), and Float Denormalized Input Flag (FDENI).

This status word represents the IEEE exceptions that were raised during the last floating-point operation. The default arrangement is that the floating-point library sets these values, but does not handle any of these exceptions.

The compiler supports these exceptions, which the IEEE standard describes:

- `IEEE_Inexact`
- `IEEE_Divide_By_Zero`
- `IEEE_Underflow`
- `IEEE_Overflow`
- `IEEE_Signaling_Nan`

The listing below is an example of how to use the exception status word.

Listing: Using the Exception Status Word

```
#include <fltmath.h>
float x,y;

. . .

x = x*y;

if (IEEE_Exceptions & IEEE_Overflow)
{

<handle overflow>

}
```

4.3.3.3 EnableFPExceptions

NOTE

EnableFPExceptions are valid only for the software floating-point implementation.

This is a bit field mask. Setting a flag enables raising an `SIGFPE` signal if the last `FP` operation raised this exception.

For example, the listing below is an example that installs a signal for handling overflow and divide-by-zero exceptions.

Listing: Setting a Signal for Exceptions

```
#include <fltmath.h>
#include <signal.h>

void SigFPHandler(int x)

{

switch (IEEE_Exceptions)
```

```
{

case IEEE_Overflow:

. . .

case IEEE_Divide_by_zero:

. . .

}

}

float x,y;

. . .

EnableFPExceptions = IEEE_Overflow | IEEE_Divide_by_zero;

signal(SIGFPE, SigFPHandler)

x = x*y /*This raises
SIGFPE if overflow or */

             divide by zero occur
*/
```

NOTE

As signal handling installs the handler address in the interrupt table, this example works only if the interrupt vector table is in RAM. If the call to `SIGNAL` cannot install the new handler, the call returns `SIG_ERR`.

4.3.3.4 Checking for floating point (-reject_floats)

To check whether an application includes any floating-point variables or operations, use the `-reject_floats` option: in the `scc` command line, include the expression

```
-scc-reject_floats
```

If the compiler front end finds any floating-point variables or operations in the application, it stops compilation and issues error messages. For example, suppose you want to apply this test to application file gamma.c. The following listing shows the command line and example error messages.

Listing: Floating Point Check

```
$ scc gamma.c -scc -reject_floats
[MWFE,2,10313,1,D:\StarCore_FrontEnd\examples\c\gamma.c:

"gamma.c", line 1: Error: illegal use of `float'

        float f = 3.14;
```

```
        ^

[MWFE,2,10313,2,D:\StarCore_FrontEnd\examples\c\gamma.c:

"gamma.c", line 2: Error: illegal use of `double'

        double d = 2.5;

        ^

[MWFE,2,10313,3,D:\StarCore_FrontEnd\examples\c\gamma.c:

"gamma.c", line 3: Error: illegal use of `double'

        long double ld = 2.52;

                ^^
```

NOTE

In response to this option, the compiler halts compilation as soon as it detects any floating-point types, even in a function header. For example, the compiler halts as soon as it sees a header such as:

```
extern void my_function(float param);
```

4.3.4 getopt function (getopt.h)

Table 4-18 lists and describes the getopt functions supported by the compiler.

Table 4-18. getopt() function

Function	Description
`getopt (int argc, char *const *argv, const char *options)`	The getopt function gets the next option argument from the argument list specified by the argv and argc arguments. Normally these values come directly from the arguments received by main. The options argument is a string that specifies the option characters that are valid for this program. An option character in this string can be followed by a colon (':') to indicate that it takes a required argument.

The following listing shows an example for getopt() function.

Listing: Example - getopt()

```
$ cat test.c
#include <stdio.h>
#include <getopt.h>

int main (int argc, char **argv)
{
     int aflag = 0, bflag = 0, index, c;
     char *cvalue = NULL;
     while ((c = getopt (argc, argv, "abc:")) != -1)
            switch (c)
```

```
            {
                case 'a':
                      aflag = 1;
                      break;
                case 'b':
                      bflag = 1;
                      break;
                case 'c':
                      cvalue = optarg;
                      break;
                case '?':
                      if (optopt == 'c')
                            fprintf (stderr, "Option -%c requires an argument.\n", optopt);
                      else
                            fprintf (stderr, "Unknown option character `\\x%x'.\n", optopt);
                      return 1;
                 default:
                      break;
               }
          printf ("aflag = %d, bflag = %d, cvalue = %s\n", aflag, bflag, cvalue);
          for (index = optind; index < argc; index++)
                printf ("Non-option argument %s\n", argv[index]);
          return 0;
}
$ scc -arch b4860 -mb -be -O4 test.c -o test.eld
$ runsim -d sc3900plat_iss test.eld
aflag = 0, bflag = 0, cvalue =
$ runsim -d sc3900plat_iss test.eld -a -b -cfoo arg1
aflag = 1, bflag = 1, cvalue = foo
Non-option argument arg1
```

4.3.5 Integer characteristics (limits.h)

Table 4-19 lists the contents of `limits.h`.

Table 4-19. Contents of file limits.h

Constant	Value	Purpose
CHAR_BIT	8	Width of `char` type, in bits
CHAR_MAX CHAR_MIN	127-128	Maximum value for `char` Minimum value for `char`
INT_MAXINT_MIN UINT_MAX	2147483647(-2147483647-1)42949 6729u	Maximum value for `int` Minimum value for `int` Maximum value for `unsigned int`
LONG_MAX LONG_MIN ULONG_MAX	2147483647(-2147483647-1)42949 6729uL	Maximum value for `long int` Minimum value for `long int` Maximum value for `unsigned long int`
MB_LEN_MAX	2	Maximum number of bytes in a multibyte character
SCHAR_MAX SCHAR_MIN UCHAR_MAX	127-128255	Maximum value for `signed char` Minimum value for `signed char` Maximum value for `unsigned char`
SHRT_MAXSHRT_MINUSHRT_MAX	32767-3276865536u	Maximum value for `short int` Minimum value for short int Maximum value for `unsigned short int`

4.3.6 Locales (locale.h)

Table 4-20 lists the locale functions that the compiler supports.

NOTE

These locale functions have no effect; the compiler supports them only for compatibility.

Table 4-20. Locale Functions

Function	Purpose
`localeconv(void)`	Not applicable
`setlocale(int category, const char* locale)`	Not applicable

4.3.7 Floating-point math (math.h)

The compiler runtime environment uses floating-point emulation to implement the `math.h` library. This library provides support for these function types:

- Trigonometric
- Hyperbolic
- Exponential and logarithmic
- Power
- Other functions

Table 4-21 lists these functions.

Table 4-21. Mathematical Functions

Type	Function	Purpose
Trigonometric	`double acos(double)`	arc cosine
	`double asin(double)`	arc sine
	`double atan(double)`	arc tangent
	`double atan2(double, double)`	arc tangent2
	`double cos(double)`	cosine
	`double sin(double)`	sine
	`double tan(double)`	tangent
Hyperbolic	`double cosh(double)`	Hyperbolic cosine

Table continues on the next page...

Table 4-21. Mathematical Functions (continued)

Type	Function	Purpose
	`double sinh(double)`	Hyperbolic sine
	`double tanh(double)`	Hyperbolic tangent
Exponential, Logarithmic	`double exp(double)`	Exponential
	`double frexp(double, int*)`	Splits floating-point into fraction and exponent
	`double ldexp(double, int)`	Computes value raised to a power
	`double log(double)`	Natural logarithm
	`double log10(double)`	Base ten (10) logarithm
	`double modf(double, double*)`	Splits floating-point into fraction and integer
Power	`double pow(double, double)`	Raises value to a power
	`double sqrt(double)`	Square root
Other	`double ceil(double)`	Ceiling
	`double fabs(double)`	Floating-point absolute number
	`double floor(double)`	Floor
	`double fmod(double, double)`	Floating-point remainder

4.3.8 Program administrative functions

Programs routinely must control jumps, handle signals, define variable arguments, and define standard types and constants. Table 4-22 lists the corresponding functions that the compiler supports, indicating their library files.

Table 4-22. Administrative Functions

Type	Function	Purpose
Non-local jumps (`setjmp.h`)	`typedef unsigned int jmp_buf[32]`	Buffer used to save the execution context
	`void longjmp(jmp_buf, int)`	Nonlocal jump
	`int setjmp(jmp_buf)`	Nonlocal return
Signal handling (`signal.h`)	`int raise(int)`	Raises a signal
	`void(*signal(int, void (*) ( int ))) (int)`	Installs a signal handler
Variable argument (s `tdarg.h`)	`va_arg(_ap, _type) (*(_type*) ((_ap) -= sizeof(_type)))`	Returns next parameter in argument list
	`va_end(_ap) (void)0`	Performs cleanup of argument list
	`va_list`	Type declaration of variable argument list

Table continues on the next page...

Table 4-22. Administrative Functions (continued)

Type	Function	Purpose
	`va_start(_ap, _parmN) (void) (_ap = (char*)&_parmN)`	Performs initialization of argument list
Standard definitions (`stddef.h`)	`NULL((void*)0)`	Null pointer constant
	`offsetof(type, member)`	Field offset in bytes from start of structure
	`typedef int ptrdiff_t`	Signed integer type resulting from the subtraction of two pointers
	`typedef int size_t`	Unsigned integer type that is the data type of the `sizeof` operator
	`typedef short wchar_t`	Wide character type, as defined in ISO C

4.3.9 I/O library (stdio.h)

The `stdio.h` library contains the input, stream, output, and miscellaneous I/O functions. Table 4-23 lists these functions.

Table 4-23. I/O functions

Type	Function	Purpose
Input	`int fgetc(FILE*)`	Inputs a single character if available from specified stream
	`size_t fread(void*, size_t, size_t, FILE*)`	Inputs a size number of characters from `stdin`
	`int fscanf(FILE*, const char*, ...)`	Inputs text from the specified stream
	`int getc(FILE*)`	Inputs a single character if available from specified stream
	`int getchar(void)`	Inputs a single character if available from `stdin`
	`int scanf(const char*, ...)`	Inputs text from `stdin`
	`int sscanf(const char*, const char*, ...)`	Inputs text from specified string
Stream	`void clearerr(FILE*)`	Clears the EOF and error indicators for the specified stream
	`int fclose(FILE*)`	Flushes the specified stream and closes the file associated with it
	`int feof(FILE*)`	Tests the EOF indicator for the specified stream
	`int ferror(FILE*)`	Tests the error indicator for the specified stream

Table continues on the next page...

Table 4-23. I/O functions (continued)

Type	Function	Purpose
	`int fgetpos(FILE*, fpos_t*)`	Stores the current value of the file position indicator for the specified stream
	`FILE* freopen(const char*,const char*,FILE*)`	Opens the specified file in the specified mode, using the specified stream
	`int fseek(FILE*, long int, int)`	Sets the file position indicator for the specified stream
	`int fsetpos(FILE*, const fpos_t*)`	Sets the file position indicator for the specified stream to the specified value
	`long int ftell(FILE*)`	Retrieves the current value of the file position indicator for the current stream
	`int remove(const char*)`	Makes the specified file unavailable by its defined name
	`int rename(const char*, const char*)`	Assigns to the specified file a new filename
	`void rewind(FILE*)`	Sets the file position indicator for the specified stream to the beginning of the file
	`void setbuf(FILE*, char*)`	Defines a buffer and associates it with the specified stream. A restricted version of `setvbuf()`
	`int setvbuf(FILE*, char*, int, size_t)`	Defines a buffer and associates it with the specified stream
	`stderr`	Standard error stream (Value = 3)
	`stdin`	Standard input stream (Value = 1)
	`stdout`	Standard output stream (Value = 2)
	`FILE* tmpfile(void)`	Creates a temporary file
	`char* tmpnam(char*)`	Generates a valid filename, meaning a filename that is not in use, as a string
Output	`char* fgets(char*, int, FILE*)`	Outputs characters to the specified stream
	`int fprintf(FILE*, const char*, ...)`	Outputs the specified text to the specified stream
	`int fputc(int, FILE*)`	Outputs a single character to the specified stream
	`int fputs(const char*, FILE*)`	Outputs a string to the specified stream
	`size_t fwrite(const void*, size_t, size_t, FILE*)`	Outputs a size number of characters to `stdout`
	`char* gets(char*)`	Outputs characters into the user's buffer
	`void perror(const char*)`	Outputs an error message
	`int printf(const char*, ...)`	Outputs the specified text to `stdout`
	`int putc(int, FILE*)`	Outputs a single character to the specified stream
	`int putchar(int)`	Outputs a single character

Table continues on the next page...

Table 4-23. I/O functions (continued)

Type	Function	Purpose
	`int puts (const char*)`	Outputs the string to `stdout`, followed by a new line
	`int sprintf(char*, const char*, ...)`	Outputs the specified text to the specified buffer
	`int vfprintf(FILE*, const char*, va_list)`	Outputs the variable arguments to the specified stream
	`int vprintf(const char*, va_list)`	Outputs the variable arguments to `stdout`
	`int vsprintf(char*, const char*, va_list)`	Outputs the variable arguments to the specified buffer
Miscellaneous	`int fflush(FILE*)`	Causes the output buffers to be emptied to their destinations
	`FILE* fopen(const char*, const char*)`	Associates a stream with a file
	`int ungetc(int, FILE*)`	Moves the character back to the head of the input stream

4.3.10 General utilities (stdlib.h)

The `stdlib.h` library contains these function types:

- Memory allocation
- Integer arithmetic
- String conversion
- Searching and sorting
- Pseudo random number generation
- Environment
- Multibyte

Table 4-24 lists these functions.

Table 4-24. General utility functions

Type	Function	Purpose
Memory allocation	`void free(void*)`	Returns allocated space to heap
	`void* calloc(size_t, size_t)`	Allocates heap space initialized to zero
	`void* malloc(size_t)`	Allocates heap space
	`void* realloc(void*, size_t)`	Allocates a larger heap space and returns previous space to heap
Integer arithmetic	`int abs(int)`	Absolute value

Table continues on the next page...

Table 4-24. General utility functions (continued)

Type	Function	Purpose
	`div_t div(int, int)`	Quotient and remainder
	`long int labs(long int)`	Computes absolute value and returns as `long int`
	`ldiv_t ldiv(long int, long int)`	Quotient and remainder of `long int`
String conversion	`double atof(const char*)`	String to float
	`int atoi(const char*)`	String to int
	`long int atol(const char*)`	Long
	`double strtod(const char*, char**)`	Double
	`long int strtol(const char*, char**, int)`	Long
	`unsigned long int strtoul(const char*, char**, int)`	Unsigned long
Searching, sorting	`void *bsearch(const void*, const void*, size_t, size_t, int(*)(const void*, const void*))`	Binary search
	`void *qsort(void*, size_t, size_t, int(*)(const void*, const void*))`	Quick sort
Pseudo random number generation	`int rand(void)`	Random number generator
	`void srand(unsigned int)`	Initializes the random number generator
Environment	`void abort(void)`	Causes an abnormal termination.
	`int atexit(void (*)(void))`	Registers a function to be called at normal termination.
	`void exit(int)`	Causes a normal termination
	`char *getenv(const char *name)`	Gets environment variable. (This function is supported for compatibility purposes and has no effect.)
	`int system(const char *string)`	Passes command to host environment. (This function is supported for compatibility purposes and has no effect.)
Multibyte character	`int mblen(const char*, size_t)`	Multibyte string length
	`size_t mbstowcs(wchar_t*, const char*, size_t)`	Converts multibyte string to wide character string
	`int mbtowc(wchar_t*, const char*, size_t)`	Converts multibyte to wide character
	`int wctomb(char*, wchar_t)`	Converts wide character to multibyte
	`size_t wcstombs (char*, const wchar_t*, size_t)`	Converts wide character string to multibyte string

4.3.11 String functions (string.h)

The `string.h` library contains these function types:

- Copying
- Concatenation
- Comparison
- Search
- Other

Table 4-25 lists these functions.

Table 4-25. String functions

Type	Function	Purpose
Copying	`void* memcpy(void*, const void*, size_t)`	Copies data
	`void* memmove(void*, const void*, size_t)`	Swaps data
	`char* strcpy(char*, const char*)`	Copies a string
	`char* strncpy(char*, const char*, size_t)`	Copies a string of a maximum length
Concatenation	`char* strcat(char*, const char*)`	Concatenates a string to the end of another string
	`char* strncat(char*, const char*, size_t)`	Concatenates a string of specified maximum length to the end of another string
Comparison	`int memcmp(const void*, const void*, size_t)`	Compares data
	`int strcmp(const char*, const char*)`	Compares strings
	`int strcoll(const char*, const char*)`	Compares strings based on locale
	`int strncmp(const char*, const char*, size_t)`	Compares strings of maximum length
	`size_t strxfrm(char*, const char*, size_t)`	Transforms a string into a second string of the specified size
Search	`void* memchr(const void*, int, size_t)`	Searches for a value in the first number of characters
	`char* strchr(const char*, int)`	Searches a string for the first occurrence of `char`
	`size_t strcspn(const char*, const char*)`	Searches a string for the first occurrence of `char` in string set and returns the number of characters skipped

Table continues on the next page...

Table 4-25. String functions (continued)

Type	Function	Purpose
	`char strpbrk(const char*, const char*)`	Searches a string for the first occurrences of `char` in string set and returns a pointer to that location
	`char* strrchr(const char*, int)`	Searches a string for the last occurrence of `char`
	`size_t strspn(const char*, const char*)`	Searches a string for the first occurrence of `char` not in string set
	`char* strstr(const char*, const char*)`	Searches a string for the first occurrence of string
	`char* strtok(char*, const char*)`	Separates a string into tokens
Other	`void* memset(void*, int, size_t)`	Copies a value into each number of characters
	`char* strerror(int)`	Returns string for associated error condition
	`size_t strlen(const char*)`	Returns size of string

NOTE

`memcpy`, `memmove`, `memset`, `memchr`, and `memcmp` are limited to 64K word blocks of memory, for example, `memset(ptr, 0x10000)` is the maximum supported value and you cannot use any value that is > `0x10000`.

4.3.12 Time functions (time.h)

Table 4-26 lists the time functions, and the time constant, that the compiler supports.

Table 4-26. Time functions

Type	Function	Purpose
Time functions (`time.h`)	`char *asctime(const struct tm *timeptr)`	Converts time to ASCII representation
	`clock_t clock()`	Returns processor time
	`typedef unsigned long clock_t`	Type used for measuring time
	`char *ctime (const time_t *timer)`	Converts time to ASCII representation
	`double difftime(time_t time1, time_t time0)`	Returns difference in seconds
	`time_t mktime(struct tm *timeptr)`	Converts `struct tm` to `time_t`

Table continues on the next page...

Table 4-26. Time functions (continued)

Type	Function	Purpose
	`size_t strftime (char *s, size_t maxsize, const char *format, const struct tm *timeptr)`	Converts an ASCII string to `time_t`
	`time_t time(time_t *timer)`	Returns processor time (same as `clock`)
	`typedef unsigned long time_t`	Type used for measuring time
	`struct tm *gmtime(const time_t *timer)`	Returns time in GMT time zone
	`struct tm *localtime(const time_t *timer)`	Returns time in local time zone
Time constant	`CLOCKS_PER_SEC`	Value is different for each target board. See the board's `time.h` file.

The `clock` function returns the current value of the system timer. This function must be configured to match the actual system timer configuration. The timer is started and set for a maximum period during the initialization of any C program that references the `clock` function, and is used only by this function. The return value of `clock` has type `clock_t`, which is `unsigned long`.

The listing below shows how to use the `clock` function to time your application.

Listing: Timing an Application

```
#include <time.h>
clock_t start, end, elapsed;

/* . . . application setup . . . */

start = clock( );

/* . . . application processing . . . */

end = clock( );

elapsed = end - start;  /* Assumes no wrap-around */

printf("Elapsed time: %Lu * 2 cycles. \n",

elapsed)
```

4.4 Calling conventions

The compiler supports a stack-based calling convention. Additional calling conventions are also supported. In addition, calling conventions can be mixed within the same application.

When compiling in separate compilation mode, non-static functions use the stack-based calling convention.

In this appendix:

4.4.1 Stack-based calling convention

For passing arguments, following registers are used:

- D0 to D7
- R0 to R7

Following calling conventions are used.

- The first (left-most) 8 parameters of pointer or integral data type are passed in R registers.
- Up to the first (left most) 8 parameters of float type or Word40 type are passed in D registers.
- Up to the first 4 parameters of (unsigned) long long or double data type are passed in pairs of D registers - D0D1, D2D3, D4D5, or D6D7.
- Up to the first 8 structures, whose size is less than 32 bits are passed in D registers.
- Up to the first 4 structures, whose size is between 32 and 64 bits are passed in pairs of D registers.
- The remaining parameters are pushed on the stack. Long parameters are pushed using the little-endian mode, with the least significant bits in the lower addresses.
- Only 8 D-registers can be used for passing arguments. The two long long and four Word40 can be passed, the following long long are passed on stack.
- The return value is returned:
  - in R0 if it has integral or pointer data type,
  - in D0 for float type, Word40 type, and structures, whose size is less than 32 bits,
  - in D0D1 for (unsigned) long long type, double type, and structures, whose size is between 32 and 64 bits.

    Functions returning large structures (structures that do not fit in a single or double register) receive and return the returned structure address in the R7 register. The space for the returned object is allocated by the caller.

- For functions with a variable number of parameters, all the fixed parameters are passed in registers (if the type permits) and all variable parameters are passed on the stack.
- Parameters are aligned in memory according to the base parameter type, with the exception of characters and unsigned characters that have a 32-bit alignment.

These registers, if actually used, are saved by the callee: D28-D31, R28-R31.

Callee should save/restore the SR bits that are modified: ASPA, W20, SM2, S, SCM, RM, SM. Default values for these SR bits are implied when entering a function.

The saturation mode is by default ON.

The listing below shows two function calls and the parameters that are allocated for each call.

Listing: Function Call and Allocation of Parameters

```
# Function call:

int alpha(int a1, struct fourbytes a2, struct eightbytes a3, int *a4)

# Parameters for the preceding function call:

# a1 - in R0

# a2 - in D0

# a3 - in D2D3

# a4 - R1

# return in R0

# Function call:

beta(long *b1, int b2, int b3[])

# Parameters for the preceding function call:

# b1 - in R0

# b2 - in R1

# b3 - in R2

# Function call:

long long gamma(Word40 c1, long long c2, fourbytes c3, int c4,
eightbytes *c5, int *c6, int c7, short c8, eightbytes c9, Word64 c10,
unsigned c11, int *c12, unsigned long long c13, short c14, int c15)

# c1 - in D0

# c2 - in D2D3

# c3 - in D1

# c4 - in R0

# c5 - in R1
```

```
# c6 - in R2

# c7 - in R3

# c8 - in R4

# c9 - in D4D5

# c10 - in D6D7

# c11 - in R5

# c12 - in R6

# c13 - on stack

# c14 - in R7

# c15 - on stack

# return in D0D1
```

The stack-based calling convention must be used when calling functions that are required to maintain a calling stack.

The compiler is able to use optimized calling sequences for functions that are not exposed to external calls.

Local and formal variables are allocated on the stack and in registers.

Table 4-27 summarizes register usage in the stack-based calling convention.

Table 4-27. Register usage in the stack-based calling convention

Register	Used as	Caller Saved	Callee Saved
D0-D7	Argument passing (long long, double, Word40, structures whose size is less than 64 bits) - D0 and/or D1 return	*	
D8-D27		*	
D28-D31			*
D32-D64		*	
R0-R7	Argument passing (numeric, pointer) - R0 return	*	
R8-R27		*	
R28-R30			*
R31	Optional frame pointer		*

4.4.2 Stack frame layout

The stack pointer points to the top (high address) of the stack frame. Space at higher addresses than the stack pointer is considered invalid and may actually be un-addressable. The stack pointer value must always be a multiple of eight.

Figure 4-1 shows typical stack frames for a function, indicating the relative position of local variables, parameters and return addresses.

Figure 4-1. Stack Frame Layout

The caller must reserve stack space for return variables that do not fit in registers. This return buffer area is typically located with the local variables. This space is typically allocated only for functions that make calls that return structures. Beyond these requirements, a function is free to manage its stack frame as necessary.

The outgoing parameter overflow block is located at the top (higher addresses) of the frame. Any incoming argument spill generated for `varargs` and `stdargs` processing must be at the bottom (low addresses) of the frame.

The caller puts argument variables that do not fit in registers into the outgoing parameter overflow area. If all arguments fit in registers, this area is not required. A caller has the option to allocate argument overflow space sufficient for the worst case call, use portions of this space as necessary, and/or leave the stack pointer unchanged between calls.

Local variables that do not fit into the local registers are allocated space in the local variables area of the stack. If there are no such variables, this area is not required .

4.4.3 Frame and argument pointers

The compiler does not use a frame pointer.

If, however, the use of a frame pointer is required by external code, R31 may be allocated as a frame pointer. When this register is allocated as frame pointer, it should be saved and restored as part of the function prolog/epilog code.

4.5 Predefined Macros

This appendix describes predefined macros available with StarCore compiler.

4.5.1 Predefined macros

A few macros are standard C macros, while others are specific to StarCore compiler and architecture.

- __COUNTER__
- __cplusplus
- __CWCC__
- __DATE__
- __embedded_cplusplus
- __FILE__
- __func__
- __FUNCTION__
- __INCLUDE_LEVEL__
- __ide_target()
- __LINE__
- __MWERKS__
- __PRETTY_FUNCTION__
- __profile__
- _SC3900FP_
- _SC3900FP_COMP_
- __SIGNED_CHARS__

4.5.1.1 __COUNTER__

Preprocessor macro that expands to an integer.

Syntax

```
__COUNTER__
```

Remarks

The compiler defines this macro as an integer that has an initial value of 0 incrementing by 1 every time the macro is used in the translation unit.

The value of this macro is stored in a precompiled header and is restored when the precompiled header is used by a translation unit.

4.5.1.2 __cplusplus

Preprocessor macro defined if compiling C++ source code.

Syntax

```
__cplusplus
```

Remarks

The compiler defines this macro when compiling C++ source code. This macro is undefined otherwise.

4.5.1.3 __CWCC__

Preprocessor macro defined as the version of the CodeWarrior compiler frontend.

Syntax

```
__CWCC__
```

Remarks

CodeWarrior compilers issued after 2006 define this macro with the compiler's frontend version. For example, if the compiler frontend version is 4.2.0, the value of `__CWCC__` is `0x4200`.

CodeWarrior compilers issued prior to 2006 used the predefined macro `__MWERKS__`. The `__MWERKS__` predefined macro is still functional as an alias for `__CWCC__`.

The ISO standards do not specify this symbol.

4.5.1.4 __DATE__

Preprocessor macro defined as the date of compilation.

Syntax

```
__DATE__
```

Remarks

The compiler defines this macro as a character string representation of the date of compilation. The format of this string is

```
"Mmm dd yyyy"
```

where *Mmm* is the a three-letter abbreviation of the month, *dd* is the day of the month, and *yyyy* is the year.

4.5.1.5 _ENTERPRISE_C_

Defined for use with the StarCore compiler. If your source file may be compiled with other compilers apart from the Freescale StarCore C compiler, this macro should be included in a conditional statement to ensure that the appropriate commands are activated, for example: `#ifdef _ENTERPRISE_C_`

(Freescale StarCore C compiler-specific commands)

```
#else

....

#endif
```

Syntax

```
_ENTERPRISE_C_
```

4.5.1.6 __embedded_cplusplus

Defined as 1 when compiling embedded C++ source code, undefined otherwise.

Syntax

```
__embedded_cplusplus
```

Remarks

The compiler defines this macro as 1 when the compiler's settings are configured to restrict the compiler to translate source code that conforms to the Embedded C++ proposed standard. The compiler does not define this macro otherwise.

4.5.1.7 __FILE__

Preprocessor macro of the name of the source code file being compiled.

Syntax

```
__FILE__
```

Remarks

The compiler defines this macro as a character string literal value of the name of the file being compiled, or the name specified in the last instance of a `#line` directive.

4.5.1.8 __func__

Predefined variable of the name of the function being compiled.

Prototype

```
static const char __func__[] = "function-name";
```

Remarks

The compiler implicitly defines this variable at the beginning of each function if the function refers to `__func__`. The character string contained by this array, *function-name*, is the name of the function being compiled.

This implicit variable is undefined outside of a function body. This variable is also undefined when C99 (ISO/IEC 9899-1999) or GCC (GNU Compiler Collection) extension settings are off.

4.5.1.9 __FUNCTION__

Predefined variable of the name of the function being compiled.

Prototype

```
static const char __FUNCTION__[] = "function-name";
```

Remarks

The compiler implicitly defines this variable at the beginning of each function if the function refers to `__FUNCTION__`. The character string contained by this array, *function-name*, is the name of the function being compiled.

This implicit variable is undefined outside of a function body.

4.5.1.10 __INCLUDE_LEVEL__

Decimal constant, indicating the current depth of file inclusion.

Syntax

```
__INCLUDE_LEVEL__
```

4.5.1.11 __ide_target()

Preprocessor operator for querying the IDE about the active build target.

Syntax

```
__ide_target("target_name")
```

target-name

The name of a build target in the active project in the CodeWarrior IDE.

Remarks

Expands to `1` if *target_name* is the same as the active build target in the CodeWarrior IDE's active project. Expands to `0` otherwise. The ISO standards do not specify this symbol.

4.5.1.12 __LINE__

Preprocessor macro of the number of the line of the source code file being compiled.

Syntax

```
__LINE__
```

Remarks

The compiler defines this macro as a integer value of the number of the line of the source code file that the compiler is translating. The `#line` directive also affects the value that this macro expands to.

4.5.1.13 __MWERKS__

Deprecated. Preprocessor macro defined as the version of the CodeWarrior compiler.

Syntax

```
__MWERKS__
```

Remarks

Replaced by the built-in preprocessor macro `__CWCC__`.

CodeWarrior compilers issued after 1995 define this macro with the compiler's version. For example, if the compiler version is 4.0, the value of `__MWERKS__` is `0x4000`.

This macro is defined as `1` if the compiler was issued before the CodeWarrior CW7 that was released in 1995.

The ISO standards do not specify this symbol.

4.5.1.14 __PRETTY_FUNCTION__

Predefined variable containing a character string of the "unmangled" name of the C++ function being compiled.

Syntax

```
static const char __PRETTY_FUNCTION__[] = "function-name";
```

Remarks

The compiler implicitly defines this variable at the beginning of each function if the function refers to `__PRETTY_FUNCTION__`. This name, *function-name*, is the same identifier that appears in source code, not the "mangled" identifier that the compiler and linker use. The C++ compiler "mangles" a function name by appending extra characters to the function's identifier to denote the function's return type and the types of its parameters.

The ISO/IEC 14882-1998 C++ standard does not specify this symbol.

4.5.1.15 __profile__

Preprocessor macro that specifies whether or not the compiler is generating object code for a profiler.

Syntax

```
__profile__
```

Remarks

Defined as 1 when generating object code that works with a profiler. Undefined otherwise. The ISO standards does not specify this symbol.

4.5.1.16 _SC3900FP_

Defined when compiling for SC3900FP/B4460/B4860.

Syntax

```
_SC3900FP_
```

4.5.1.17 _SC3900FP_COMP_

Defined in order to use intrinsics definitions for the SC3900 architecture with the SC3900FP intrinsic definitions.

Syntax

```
_SC3900FP_COMP_
```

Example

`void __ash_lft_2x_da` declaration is different for SC3900 and SC3900FP architectures.

For SC3900FP compiler:

```
__ash_lft_2x_da(Word32 __Da, Word40 __Db, Word40 __Dc, Word40* __Dd,
Word40* __De)
```

For SC3900 compiler:

```
void __ash_lft_2x(Word32 __Da, Word40 __Db, Word40 __Dc, Word40* __Dd, Word40* __De)
```

By defining `_SC3900FP_COMP_`, `__ash_lft_2x_da(Word32 __Da, Word40 __Db, Word40 __Dc, Word40* __Dd, Word40* __De)` can be used with the SC3900 compiler as well.

4.5.1.18 __SIGNED_CHARS__

Defined when char is signed by default.

Syntax

```
__SIGNED_CHARS__
```

4.5.1.19 __STDC__

Defined as 1 when compiling ISO/IEC Standard C source code, undefined otherwise.

Syntax

```
__STDC__
```

Remarks

The compiler defines this macro as 1 when the compiler's settings are configured to restrict the compiler to translate source code that conforms to the ISO/IEC 9899-1990 and ISO/IEC 9899-1999 standards. The compiler does not define this macro otherwise.

4.5.1.20 __STDC_VERSION__

Defined in ANSI C mode as 199409L.

Syntax

```
__STDC_VERSION__
```

4.5.1.21 _SOFTFPA_

Distinguishes between software and hardware floating point arithmetic.

Syntax

```
_SOFTFPA_
```

4.5.1.22 __TIME__

Preprocessor macro defined as a character string representation of the time of compilation.

Syntax

```
__TIME__
```

Remarks

The compiler defines this macro as a character string representation of the time of compilation. The format of this string is

```
"hh:mm:ss"
```

where *hh* is a 2-digit hour of the day, *mm* is a 2-digit minute of the hour, and *ss* is a 2-digit second of the minute.

4.5.1.23 __VERSION__

The version number of the compiler, as a character string in the form `nn.nn`.

Syntax

```
__VERSION__
```

4.5.1.24 _RENAME__LOG2_AS___LOG2_

Defined in order to use the `__log2` intrinsic/emulation call as `___log2` so as to avoid conflict with third-party definitions of `__log2`.

Syntax

```
_RENAME__LOG2_AS___LOG2_
```

4.6 C++ specific features

This appendix lists the C++ language specific features in the StarCore compiler.

In this appendix:

- C++ specific command-line options
- Extensions to standard C++
- Implementation-defined behavior
- GCC extensions
- C++ specific pragmas

4.6.1 C++ specific command-line options

This section lists the C++ command-line options:

- -bool
- -Cpp_exceptions
- -gccincludes
- -RTTI
- -wchar_t

4.6.1.1 -bool

Controls the use of `true` and `false` keywords for the C++ `bool` data type.

Syntax

```
-bool on | off
```

Options

`on`

Enabled by default. The compiler recognizes `true` and `false` keywords for the `bool` data type.

`off`

The compiler does not recognize `true` and `false` keywords for the `bool` data type.

4.6.1.2 -Cpp_exceptions

Controls the use of exceptions in C++ source files.

Syntax

```
-Cpp_exceptions on | off
```

Options

`on`

Enabled by default. The compiler recognizes `try`, `catch`, and `throw` keywords, and generates additional executable code and data that performs C++ exception handling.

`off`

The compiler does not perform C++ exception handling.

4.6.1.3 -gccincludes

Controls the compilers use of GCC `#include` semantics.

Syntax

```
-gccinc[ludes]
```

Remarks

Use `-gccincludes` to control the StarCore compiler's understanding of GNU Compiler Collection (GCC) semantics. When enabled, the semantics:

- add `-I-` paths to the systems list if `-I-` is not already specified
- search referencing file's directory first for `#include` files (same as `-cwd include`). The compiler only searches the access paths, and do not take the currently `#include` file into account.

This command is global.

4.6.1.4 -RTTI

Controls the availability of runtime type information (RTTI).

Syntax

```
-RTTI on | off
```

Remarks

Default setting is `on`.

4.6.1.5 -wchar_t

Controls the use of the `wchar_t` data type in C++ source code.

Syntax

```
-wchar_t on | off
```

Remarks

The `-wchar on` option instructs the C++ compiler to recognize the `wchar_t` type as a built-in type for the wide characters. The `-wchar off` option tells the compiler not to allow this built-in type, forcing the user to provide a definition for this type. Default setting is `on`.

4.6.2 Extensions to standard C++

The StarCore C++ compiler has following features and capabilities that are not described in the ISO/IEC 14882:2003 C++ Standard:

4.6.2.1 __PRETTY_FUNCTION__ Identifier

The `__PRETTY_FUNCTION__` predefined identifier is an alternative for `__FUNC__` (a predefined variable that expands to the current function name). The identifier represents the qualified (un-mangled) C++ name of the function being compiled.

4.6.2.2 Standard and non-standard template parsing

The StarCore C++ compiler has options that let you specify how strictly template declarations and instantiations are translated. When using its strict template parser, the compiler expects the `typename` and `template` keywords to qualify names, preventing the same name in different scopes or overloaded declarations from being inadvertently used. When using its regular template parser, the compiler makes guesses about names in templates, but may guess incorrectly about which name to use.

A qualified name that refers to a type and that depends on a template parameter must begin with `typename` (according to the ISO/IEC 14882:2003 C++ Standard). The listing below shows an example.

Listing: Using the typename keyword

```
template <typename T> void f()
{
  T::name *ptr; // ERROR: an attempt to multiply T::name by ptr
  typename T::name *ptr; // OK
}
```

The compiler requires the `template` keyword at the end of "`.`" and "`->`" operators, and for qualified identifiers that depend on a template parameter. The listing below shows an example.

Listing: Using the template keyword

```
template <typename T> void f(T* ptr)
{
  ptr->f<int>(); // ERROR: f is less than int
  ptr->template f<int>(); // OK
}
```

Names referred to inside a template declaration that are not dependent on the template declaration (that do not rely on template arguments) must be declared before the template's declaration. These names are bound to the template declaration at the point where the template is defined. Bindings are not affected by definitions that are in scope at the point of instantiation. The listing below shows an example.

Listing: Binding non-dependent identifiers

```
void f(char);

template <typename T> void tmpl_func()
{
  f(1); // Uses f(char); f(int), below, is not defined yet.
  g(); // ERROR: g() is not defined yet.
}
void g();
void f(int);
```

Names of template arguments that are dependent in base classes must be explicitly qualified (according to the ISO/IEC 14882:2003 C++ Standard). See the below listing.

Listing: Qualifying template arguments in base classes

```
template <typename T> struct Base
{
  void f();
}
template <typename T> struct Derive: Base<T>
{
  void g()
  {
    f(); // ERROR: Base<T>::f() is not visible.
    Base<T>::f(); // OK
  }
}
```

When a template contains a function call in which at least one of the function's arguments is type-dependent, the compiler uses the name of the function in the context of the template definition (according to the ISO/IEC 14882:2003 C++ Standard) and the context of its instantiation (according to the ISO/IEC 14882:2003 C++ Standard). The listing below shows an example.

Listing: Function call with type-dependent argument

```
void f(char);

template <typename T> void type_dep_func()
{
  f(1); // Uses f(char), above; f(int) is not declared yet.
  f(T()); // f() called with a type-dependent argument.
}

void f(int);
struct A{};
void f(A);

int main()
{
  type_dep_func<int>(); // Calls f(char) twice.
  type_dep_func<A>(); // Calls f(char) and f(A);
```

```
  return 0;
}
```

The compiler only uses external names to look up type-dependent arguments in function calls. See the below listing.

Listing: Function call with type-dependent argument and external names

```
static void f(int); // f() is internal.

template <typename T> void type_dep_fun_ext()
{
  f(T()); // f() called with a type-dependent argument.
}

int main()
{
  type_dep_fun_ext<int>(); // ERROR: f(int) must be external.
}
```

The compiler does not allow expressions in inline assembly statements that depend on template parameters. See the below listing.

Listing: Assembly statements cannot depend on template arguments

```
template <typename T> void asm_tmpl()
{
  asm { move #sizeof(T), D0 ); // ERROR: Not supported.
}
```

The compiler also supports the address of template-id rules. See the below listing.

Listing: Address of Template-id Supported

```
template <typename T> void funcA(T) {}
template <typename T> void funcB(T) {}
...
funcA{ &funcB<int> );     // now accepted
```

4.6.3 Implementation-defined behavior

The StarCore C++ compiler implements behavior quantities listed in Table 4-28, based on the ISO/IEC 14882:2003 C++ Standard.

NOTE

The term *unlimited* in Table 4-28 means that a behavior is limited only by the processing speed or memory capacity of the computer on which the StarCore C++ compiler is running.

Table 4-28. Implementation Quantities for C/C++ Compiler (ISO/IEC 14882:2003 C++)

Behavior	Standard Minimum Guideline	CodeWarrior Limit
Nesting levels of compound statements, iteration control structures, and selection control structures	256	Unlimited
Nesting levels of conditional inclusion	256	256
Pointer, array, and function declarators (in any combination) modifying an arithmetic, structure, union, or incomplete type in a declaration	256	Unlimited
Nesting levels of parenthesized expressions within a full expression	256	Unlimited
Number of initial characters in an internal identifier or macro name	1024	Unlimited
Number of initial characters in an external identifier	1024	Unlimited
External identifiers in one translation unit	65536	Unlimited
Identifiers with block scope declared in one block	1024	Unlimited
Macro identifiers simultaneously defined in one translation unit	65536	Unlimited
Parameters in one function definition	256	Unlimited
Arguments in one function call	256	Unlimited
Parameters in one macro definition	256	256
Arguments in one macro invocation	256	256
Characters in one logical source line	65536	Unlimited
Characters in a character string literal or wide string literal (after concatenation)	65536	Unlimited
Size of an object	262144	2 GB
Nesting levels for `# include` files	256	256
`Case` labels for a `switch` statement (excluding those for any nested `switch` statements)	16384	Unlimited
Data members in a single class, structure, or union	16384	Unlimited
Enumeration constants in a single enumeration	4096	Unlimited
Levels of nested class, structure, or union definitions in a single struct-declaration-list	256	Unlimited
Functions registered by `atexit()`	32	64
Direct and indirect base classes	16384	Unlimited

Table continues on the next page...

Table 4-28. Implementation Quantities for C/C++ Compiler (ISO/IEC 14882:2003 C++) (continued)

Behavior	Standard Minimum Guideline	CodeWarrior Limit
Direct base classes for a single class	1024	Unlimited
Members declared in a single class	4096	Unlimited
Final overriding virtual functions in a class, accessible or not	16384	Unlimited
Direct and indirect virtual bases of a class	1024	Unlimited
Static members of a class	1024	Unlimited
Friend declarations in a class	4096	Unlimited
Access control declarations in a class	4096	Unlimited
Member initializers in a constructor definition	6144	Unlimited
Scope qualifications of one identifier	256	Unlimited
Nested external specifications	1024	Unlimited
Template arguments in a template declaration	1024	Unlimited
Recursively nested template instantiations	17	64 (adjustable up to 30000 using #pragma template_depth(<n>))
Handlers per try block	256	Unlimited
Throw specifications on a single function declaration	256	Unlimited

4.6.4 GCC extensions

The StarCore C++ compiler recognizes some extensions to the ISO/IEC 14882:2003 C++ Standard that are also recognized by the GCC (GNU Compiler Collection) C++ compiler.

The compiler allows the use of the :: operator, of the form *class*::*member*, in a class declaration.

Listing: Using the :: operator in class declarations

```
class MyClass {
   int MyClass::getval();
};
```

4.6.5 C++ specific pragmas

The C++ specific pragmas are:

- access_errors
- arg_dep_lookup
- ARM_conform
- ARM_scoping
- array_new_delete
- bool
- cplusplus
- cpp_extensions
- debuginline
- def_inherited
- defer_codegen
- defer_defarg_parsing
- direct_destruction
- ecplusplus
- exceptions
- extended_errorcheck
- iso_templates
- new_mangler
- no_conststringconv
- no_static_dtors
- nosyminline
- old_friend_lookup
- old_pods
- opt_classresults
- RTTI
- suppress_init_code
- template_depth
- thread_safe_init
- warn_hidevirtual
- warn_no_explicit_virtual
- warn_structclass
- wchar_type

4.6.5.1 access_errors

Controls whether to display an error message instead of a warning message in case of invalid access to protected or private class members.

Syntax

```
#pragma access_errors on | off | reset
```

Remarks

If you enable this pragma, the compiler issues an error message instead of a warning message when it detects invalid access to protected or private class members.

This pragma does not correspond to any IDE panel setting. By default, this pragma is `on`.

4.6.5.2 arg_dep_lookup

Controls C++ argument-dependent name lookup.

Syntax

```
#pragma arg_dep_lookup on | off | reset
```

Remarks

If you enable this pragma, the C++ compiler uses argument-dependent name lookup.

This pragma does not correspond to any IDE panel setting. By default, this setting is `on`.

4.6.5.3 ARM_conform

This pragma is no longer available. Use `ARM_scoping` instead.

4.6.5.4 ARM_scoping

Controls the scope of variables declared in the expression parts of `if`, `while`, `do`, and `for` statements.

Syntax

```
#pragma ARM_scoping on | off | reset
```

Remarks

If you enable this pragma, any variables you define in the conditional expression of an `if`, `while`, `do`, or `for` statement remain in scope until the end of the block that contains the statement. Otherwise, the variables only remain in scope until the end of that statement. The listing below shows an example.

By default, this pragma is `off`.

Listing: Example of Using Variables Declared in for Statement

```
for(int i=1; i<1000; i++) { /* . . . */ }
return i;  // OK if ARM_scoping is on, error if ARM_scoping is off.
```

4.6.5.5 array_new_delete

Enables the operator `new[]` and `delete[]` in array allocation and deallocation operations, respectively.

Syntax

```
#pragma array_new_delete on | off | reset
```

Remarks

By default, this pragma is `on`.

4.6.5.6 bool

Determines whether or not `bool`, `true`, and `false` are treated as keywords in C++ source code.

Syntax

```
#pragma bool on | off | reset
```

Remarks

If you enable this pragma, you can use the standard C++ `bool` type to represent `true` and `false`. Disable this pragma if `bool`, `true`, or `false` are defined in your source code.

Enabling the `bool` data type and its `true` and `false` values is not equivalent to defining them in source code with `typedef`, `enum`, or `#define`. The C++ `bool` type is a distinct type defined by the ISO/IEC 14882:2003 C++ Standard. Source code that does not treat `bool` as a distinct type might not compile properly.

By default, this setting is `on`.

4.6.5.7 cplusplus

Controls whether or not to translate subsequent source code as C or C++ source code.

Syntax

```
#pragma cplusplus on | off | reset
```

Remarks

If you enable this pragma, the compiler translates the source code that follows as C++ code. Otherwise, the compiler uses the suffix of the filename to determine how to compile it. If a file name ends in `.c`, `.h`, or `.pch`, the compiler automatically compiles it as C code, otherwise as C++. Use this pragma only if a file contains both C and C++ code.

NOTE

The StarCore C/C++ compilers do not distinguish between uppercase and lowercase letters in file names and file name extensions except on UNIX-based systems.

By default, this pragma is disabled.

Using the CodeWarrior IDE, follow these steps to enable `cplusplus` pragma:

1. Start the IDE.
2. In the **CodeWarrior Projects** view, select the project for which you want to modify the build properties.
3. Select **Project** > **Properties**.

   The **Properties for** <project> window appears. The left side of this window has a Properties list. This list shows the build properties that apply to the current project.

4. Expand the C/C++ **Build** property.
5. Select **Settings**.
6. Use the **Configuration** drop-down list to specify the launch configuration for which you want to modify the build properties.
7. Click the **Tool Settings** tab.

   The corresponding page comes forward.

8. From the list of tools on the **Tool Settings** page, select **StarCore Environment** .

   The corresponding page appears.

9. Check the **Force C++ Compilation** checkbox.
10. Click **Apply**.

For more information on compiling C++ source code, refer to the topic How to compile C++ source files.

4.6.5.8 cpp_extensions

Controls language extensions according to the ISO/IEC 14882:2003 C++ Standard.

Syntax

```
pragma cpp_extensions on | off | reset
```

Remarks

If you enable this pragma, you can use the following extensions to the ISO/IEC 14882:2003 C++ Standard that would otherwise be invalid:

- Anonymous `struct` & `union` objects. The listing below shows an example.
  Listing: Example of Anonymous struct & union Objects

```
#pragma cpp_extensions on
void func()
{
  union {
    long  hilo;
    struct { short hi, lo; }; //  anonymous struct
  };
  hi=0x1234;
  lo=0x5678;  //  hilo==0x12345678
}
```

- Unqualified pointer to a member function. The listing below shows an example.
  Listing: Example of an Unqualified Pointer to a Member Function

```
#pragma cpp_extensions on
struct RecA { void f(); }
void RecA::f()
{
  void (RecA::*ptmf1)() = &RecA::f; // ALWAYS OK

  void (RecA::*ptmf2)() = f; // OK if you enable cpp_extensions.
}
```

- Inclusion of `const` data in precompiled headers.

By default, this pragma is disabled.

4.6.5.9 debuginline

Controls whether the compiler emits debugging information for expanded inline function calls.

Syntax

```
#pragma debuginline on | off | reset
```

Remarks

If the compiler emits debugging information for inline function calls, then the debugger can step to the body of the inlined function. This behavior more closely resembles the debugging experience for un-inlined code.

NOTE

Since the actual "call" and "return" instructions are no longer present when stepping through inline code, the debugger will immediately jump to the body of an inlined function and "return" before reaching the return statement for the function. Thus, the debugging experience of inlined functions may not be as smooth as debugging un-inlined code.

This pragma does not correspond to any panel setting. By default, this pragma is `on`.

4.6.5.10 def_inherited

Controls the use of `inherited`.

Syntax

```
#pragma def_inherited on | off | reset
```

Remarks

The use of this pragma is deprecated. It lets you use the non-standard `inherited` symbol in C++ programming by implicitly adding

```
typedef base inherited;
```

as the first member in classes with a single base class.

NOTE

The ISO/IEC 14882:2003 C++ Standard does not support the `inherited` symbol. Only the StarCore C++ compiler supports the `inherited` symbol for single inheritance.

By default, this pragma is `off`.

4.6.5.11 defer_codegen

Obsolete pragma. Replaced by interprocedural analysis option.

4.6.5.12 defer_defarg_parsing

Defers the parsing of default arguments in member functions.

Syntax

```
#pragma defer_defarg_parsing on | off
```

Remarks

To be accepted as valid, some default expressions with template arguments will require additional parenthesis. For example, the listing below results in an error message.

Listing: Deferring parsing of default arguments

```
template<typename T,typename U> struct X { T t; U u; };

struct Y {
   // The following line is not accepted, and generates
   // an error message with defer_defarg_parsing on.
   void f(X<int,int> = X<int,int>());
};
```

The listing below does not generate an error message.

Listing: Correct default argument deferral

```
template<typename T,typename U> struct X { T t; U u; };

struct Y {
   // The following line is OK if the default
   // argument is parenthesized.
   void f(X<int,int> = (X<int,int>()) );
};
```

This pragma does not correspond to any panel setting. By default, this pragma is `on`.

4.6.5.13 direct_destruction

This pragma is obsolete. It is no longer available.

4.6.5.14 ecplusplus

Controls the use of embedded C++ features.

Syntax

```
#pragma ecplusplus on | off | reset
```

Remarks

If you enable this pragma, the C++ compiler disables the non-EC++ features of the ISO/IEC 14882:2003 C++ Standard, such as templates, multiple inheritance, and so on.

By default, this pragma is `off`.

4.6.5.15 exceptions

Controls the availability of C++ exception handling.

Syntax

```
#pragma exceptions on | off | reset
```

Remarks

If you enable this pragma, you can use the `try` and `catch` statements in C++ to perform exception handling. If your program does not use exception handling, disable this setting to make your program smaller.

You can throw exceptions across any code compiled by the StarCore C/C++ compiler with `#pragma exceptions on`.

You cannot throw exceptions across libraries compiled with `#pragma exceptions off`. If you throw an exception across such a library, the code calls `terminate()` and exits.

By default, this pragma is `on`.

4.6.5.16 extended_errorcheck

Controls the warning messages for possible unintended logical errors.

Syntax

```
#pragma extended_errorcheck on | off | reset
```

Remarks

If you enable this pragma, the C++ compiler generates a warning message for the possible unintended logical errors.

It also issues a warning message when it encounters a delete operator for a class or structure that has not been defined yet. The listing below shows an example.

Listing: Attempting to delete an undefined structure

```
#pragma extended_errorcheck on
struct X;
int func(X *xp)
{
     delete xp;     // Warning: deleting incomplete type X
}
```

An empty `return` statement in a function that is not declared `void`. For example, the listing below results in a warning message.

Listing: A non-void function with an empty return statement

```
int MyInit(void)
{
  int err = GetMyResources();
  if (err != -1)
  {
    err = GetMoreResources();
  }
  return; /* WARNING: empty return statement */
}
```

The listing below shows how to prevent this warning message.

Listing: A non-void function with a proper return statement

```
int MyInit(void)
{
  int err = GetMyResources();
  if (err != -1)
  {
    err = GetMoreResources();
  }
  return err; /* OK */
}
```

By default, this setting is `off`.

4.6.5.17 iso_templates

Controls whether to use the new parser supported by the StarCore C++ compiler, and issues appropriate warning messages for missing typenames.

Syntax

```
#pragma iso_templates on | off | reset
```

Remarks

This pragma ensures that your C++ source code is compiled using the newest version of the parser, which is stricter than earlier versions. The compiler issues a warning message if a typenames required by the C++ standard is missing but can still be determined by the compiler based on the context of the surrounding C++ syntax.

By default, this pragma is `on`.

4.6.5.18 new_mangler

Controls the inclusion or exclusion of a template instance's function return type to the mangled name of the instance.

Syntax

```
#pragma new_mangler on | off | reset
```

Remarks

The C++ standard requires that the function return type of a template instance to be included in the mangled name, which can cause incompatibilities. Enabling this pragma within a prefix file resolves those incompatibilities.

This pragma does not correspond to any panel setting. By default, this pragma is `on`.

4.6.5.19 no_conststringconv

Disables the deprecated implicit const string literal conversion (according to the ISO/IEC 14882:2003 C++ Standard).

Syntax

```
#pragma no_conststringconv on | off | reset
```

Remarks

When enabled, the compiler generates an error message when it encounters an implicit const string conversion.

Listing: Example of const string conversion

```
#pragma no_conststringconv on

char *cp = "Hello World"; /* Generates an error message. */
```

This pragma does not correspond to any panel setting. By default, this pragma is `off`.

4.6.5.20 no_static_dtors

Controls the generation of static destructors in C++.

Syntax

```
#pragma no_static_dtors on | off | reset
```

Remarks

If you enable this pragma, the compiler does not generate destructor calls for static data objects. Use this pragma to generate smaller object code for C++ programs that never exit (and consequently never need to call destructors for static objects).

This pragma does not correspond to any panel setting. By default, this setting is disabled.

4.6.5.21 nosyminline

Controls whether debug information is gathered for inline/template functions.

Syntax

```
#pragma nosyminline on | off | reset
```

Remarks

When on, debug information is not gathered for inline/template functions.

This pragma does not correspond to any panel setting. By default, this pragma is disabled.

4.6.5.22 old_friend_lookup

Implements non-standard C++ friend declaration behavior that allows friend declarations to be visible in the enclosing scope.

```
#pragma old_friend_lookup on | off | reset
```

Example

This example shows friend declarations that are invalid without `#pragma old_friend_lookup`.

Listing: Valid and invalid declarations without #pragma old_friend_lookup

```
class C2;
void f2();

struct S {

     friend class C1;

     friend class C2;

     friend void f1();

     friend void f2();

};

C1 *cp1;    // error, C1 is not visible without namespace declaration

C2 *cp2;    // OK

int main()

{

     f1();  // error, f1() is not visible without namespace declaration

     f2();  // OK

}
```

4.6.5.23 old_pods

Permits non-standard handling of classes, structs, and unions containing pointer-to-pointer members

Syntax

```
#pragma old_pods on | off | reset
```

Remarks

According to the ISO/IEC 14882:2003 C++ Standard, classes/structs/unions that contain pointer-to-pointer members are now considered to be plain old data (POD) types.

This pragma can be used to get the old behavior.

4.6.5.24 opt_classresults

Controls the omission of the copy constructor call for class return types if all return statements in a function return the same local class object.

Syntax

```
#pragma opt_classresults on | off | reset
```

Remarks

The listing below shows an example.

Listing: Example #pragma opt_classresults

```
#pragma opt_classresults on

struct X {
  X();
  X(const X&);
  // ...
};

X f() {
  X x; // Object x will be constructed in function result buffer.
  // ...
  return x; // Copy constructor is not called.
}
```

This pragma does not correspond to any panel setting. By default, this pragma is `on`.

4.6.5.25 RTTI

Controls the availability of runtime type information.

Syntax

```
#pragma RTTI on | off | reset
```

Remarks

If you enable this pragma, you can use runtime type information (or RTTI) features such as `dynamic_cast` and `typeid`. The other RTTI expressions are available even if you disable this pragma. Note that `*type_info::before(const type_info&)` is not implemented.

4.6.5.26 suppress_init_code

Controls the suppression of static initialization object code.

Syntax

```
#pragma suppress_init_code on | off | reset
```

Remarks

If you enable this pragma, the compiler does not generate any code for static data initialization such as C++ constructors.

WARNING

Using this pragma can produce erratic or unpredictable behavior in your program.

This pragma does not correspond to any panel setting. By default, this pragma is disabled.

4.6.5.27 template_depth

Controls how many nested or recursive class templates you can instantiate.

```
#pragma template_depth(n)
```

Remarks

This pragma lets you increase the number of nested or recursive class template instantiations allowed. By default, n equals 64; it can be set from 1 to 30000. You should always use the default value unless you receive the error message

```
template too complex or recursive
```

This pragma does not correspond to any panel setting.

4.6.5.28 thread_safe_init

Controls the addition of extra code in the binary to ensure that multiple threads cannot enter a static local initialization at the same time.

Syntax

```
#pragma thread_safe_init on | off | reset
```

Remarks

A C++ program that uses multiple threads and static local initializations introduces the possibility of contention over which thread initializes static local variable first. When the pragma is `on`, the compiler inserts calls to mutex functions around each static local initialization to avoid this problem. The C++ runtime library provides these mutex functions.

Listing: Static local initialization example

```
int func(void) {
  // There may be synchronization problems if this function is
  // called by multiple threads.
  static int countdown = 20;

  return countdown--;
}
```

NOTE

This pragma requires runtime library functions which may not be implemented on all platforms, due to the possible need for operating system support.

The listing below shows another example.

Listing: Example thread_safe_init

```
#pragma thread_safe_init on

void thread_heavy_func()
{
  // Multiple threads can now safely call this function:
  // the static local variable will be constructed only once.
  static std::string localstring = thread_unsafe_func();
}
```

NOTE

When an exception is thrown from a static local initializer, the initializer is retried by the next client that enters the scope of the local.

This pragma does not correspond to any panel setting. By default, this pragma is `off`.

4.6.5.29 warn_hidevirtual

Controls the recognition of a non-virtual member function that hides a virtual function in a superclass.

Syntax

```
#pragma warn_hidevirtual on | off | reset
```

Remarks

If you enable this pragma, the compiler issues a warning message if you declare a non-virtual member function that hides a virtual function in a superclass. One function hides another if it has the same name but a different argument type. The listing below shows an example.

Listing: Hidden Virtual Functions

```
class A {
  public:
    virtual void f(int);
    virtual void g(int);
};

class B: public A {
  public:
    void f(char);        // WARNING: Hides A::f(int)
    virtual void g(int); // OK: Overrides A::g(int)
};
```

The ISO/IEC 14882:2003 C++ Standard does not require this pragma.

NOTE

A warning message normally indicates that the pragma name is not recognized, but an error indicates either a syntax problem or that the pragma is not valid in the given context.

4.6.5.30 warn_no_explicit_virtual

Controls the issuing of warning messages if an overriding function is not declared with a virtual keyword.

Syntax

```
#pragma warn_no_explicit_virtual on | off | reset
```

Remarks

The listing below shows an example.

Listing: Example of warn_no_explicit_virtual pragma

```
#pragma warn_no_explicit_virtual on

struct A {
  virtual void f();
};

struct B {
  void f();
  // WARNING: override B::f() is declared without virtual keyword
}
```

Tip

This warning message is not required by the ISO/IEC 14882:2003 C++ Standard, but can help you track down unwanted overrides.

This pragma does not correspond to any panel setting. By default, this pragma is `off`.

4.6.5.31 warn_structclass

Controls the issuing of warning messages for the inconsistent use of the `class` and `struct` keywords.

Syntax

```
#pragma warn_structclass on | off | reset
```

Remarks

If you enable this pragma, the compiler issues a warning message if you use the `class` and `struct` keywords in the definition and declaration of the same identifier.

Listing: Inconsistent use of class and struct

```
class X;
struct X { int a; }; // WARNING
```

Use this warning when using static or dynamic libraries to link with object code produced by another C++ compiler that distinguishes between class and structure variables in its name " mangling."

By default, this pragma is disabled.

4.6.5.32 wchar_type

Controls the availability of the `wchar_t` data type in C++ source code.

Syntax

```
#pragma wchar_type on | off | reset
```

Remarks

If you enable this pragma, `wchar_t` is treated as a built-in type. Otherwise, the compiler does not recognize this type.

By default, this pragma is enabled.

4.7 Intrinsics supported in emulation and MEX library

This appendix lists the intrinsics supported in the Emulation and MEX libraries.

4.7.1 Intrinsics supported in emulation library

The following table lists such intrinsics.

Table 4-29. Intrinsics Supported in Emulation Library

Intrinsics Supported in Emulation Library			
abs_2t	abs_2w	abs_2x	abs_4t
abs_4w	abs_l	abs_leg_x	abs_s_2w
abs_s_2x	abs_s_4w	abs_s_l	abs_s_x
abs_s40_x	abs_x	absa	add_2t
add_leg_x	add_s_x	add_x	adda_lin
addla_1_lin	addla_2_lin	addla_3_lin	addla_4_lin
addla_5_lin	addla_6_lin	addm_hl_2x	addm_hl_x
addm_x_h	addm_x_l	divp_0	divp_2
divp_3	l_abs_2t	l_abs_2x	l_abs_s_2x
l_abs_s40_x	l_add_2t	l_add_leg_x	l_add_s_x
l_add_t_hh	l_add_t_hl	l_add_t_lh	l_add_t_ll
l_add_x	l_addm_hl_2x	l_addm_hl_x	l_addm_x_h
l_addm_x_l	l_divp_0	l_divp_1	l_divp_2
l_divp_3	fabs_sp	fabs_2sp	fadd_sp
fadd_2sp	fadd_x_2sp	faddsub_2sp	faddsub_x_2sp
finvsqrt_pre	finvsqrt_2sp	fix2flt_l_sp	fix2flt_l_2sp
fix2flt_ul_sp	fix2flt_ul_2sp	flog2	flog2_pre1
flog2_pre2	flt2fix_sp_l	flt2fix_sp_x	l_flt2fix_sp_x
flt2fix_sp_2x	l_flt2fix_sp_2x	fmadd_sp	fmadd_2sp
fmadd_invsq_sp	fmadd_log2_sp	fmadd_recip_sp	fmpy_sp
fmpy_2sp	fmpy_nn_2sp	fmpy_nnx_2sp	fmpy_np_2sp
fmpy_npx_2sp	fmpy_pn_2sp	fmpy_pnx_2sp	fmpy_pp_2sp
fmsod_aai_2sp	fmsod_asi_2sp	fmsod_asx_2sp	fmsod_sai_2sp
fmsod_sax_2sp	fmsod_ssi_2sp	fmsub_sp	fmsub_2sp
frecip	frecip_pre	fsub_sp	fsub_2sp
fsub_x_2sp	fsubadd_2sp	fsubadd_x_2sp	l_max_2x
l_max_x	l_maxm_2x	l_maxm_x	l_min_2x
l_min_x	l_neg_2t	l_neg_2x	l_neg_4t
l_neg_leg_x	l_neg_s_2x	l_neg_x	l_sod_aaaai_4t

Table continues on the next page...

Table 4-29. Intrinsics Supported in Emulation Library (continued)

Intrinsics Supported in Emulation Library			
l_sod_aaaax_4t	l_sod_aaii_2t	l_sod_aaii_2x	l_sod_aaii_s_2x
l_sod_aassi_4t	l_sod_aaxx_2t	l_sod_aaxx_2x	l_sod_aaxx_s_2x
l_sod_as_2x	l_sod_as_s_2x	l_sod_asasi_4t	l_sod_asasx_4t
l_sod_asii_2t	l_sod_asii_2x	l_sod_asii_s_2x	l_sod_asxx_2t
l_sod_asxx_2x	l_sod_asxx_s_2x	l_sod_sa_2x	l_sod_sa_s_2x
l_sod_saii_2t	l_sod_saii_2x	l_sod_saii_s_2x	l_sod_sasai_4t
l_sod_sasax_4t	l_sod_saxx_2t	l_sod_saxx_2x	l_sod_saxx_s_2x
l_sod_ssaai_4t	l_sod_ssii_2t	l_sod_ssii_2x	l_sod_ssii_s_2x
l_sod_ssssi_4t	l_sod_ssssx_4t	l_sod_ssxx_2t	l_sod_ssxx_2x
l_sod_ssxx_s_2x	l_sodhl_asas_4t	l_sodhl_sasa_4t	l_sub_2t
l_sub_leg_x	l_sub_s_x	l_sub_t_hh	l_sub_t_hl
l_sub_t_lh	l_sub_t_ll	l_sub_x	max_2t
max_2w	max_2x	max_4t	max_4w
max_x	maxa	maxm_2x	maxm_x
min_2t	min_2w	min_2x	min_4t
min_4w	min_x	mina	neg_2t
neg_2x	neg_4t	neg_leg_x	neg_s_2w
neg_s_2x	neg_s_4w	neg_s_l	neg_s_x
neg_x	nega	sod_aaaai_4t	sod_aaaai_s_4w
sod_aaaax_4t	sod_aaaax_s_4w	sod_aaii_2t	sod_aaii_2x
sod_aaii_s_2w	sod_aaii_s_2x	sod_aassi_4t	sod_aassi_s_4w
sod_aaxx_2t	sod_aaxx_2x	sod_aaxx_s_2w	sod_aaxx_s_2x
sod_as_2x	sod_as_s_2x	sod_asasi_4t	sod_asasi_s_4w
sod_asasx_4t	sod_asasx_s_4w	sod_asii_2t	sod_asii_2x
sod_asii_s_2w	sod_asii_s_2x	sod_asxx_2t	sod_asxx_2x
sod_asxx_s_2w	sod_asxx_s_2x	sod_sa_2x	sod_sa_s_2x
sod_saii_2t	sod_saii_2x	sod_saii_s_2w	sod_saii_s_2x
sod_sasai_4t	sod_sasai_s_4w	sod_sasax_4t	sod_sasax_s_4w
sod_saxx_2t	sod_saxx_2x	sod_saxx_s_2w	sod_saxx_s_2x
sod_ssaai_4t	sod_ssaai_s_4w	sod_ssii_2t	sod_ssii_2x
sod_ssii_s_2w	sod_ssii_s_2x	sod_ssssi_4t	sod_ssssi_s_4w
sod_ssssx_4t	sod_ssssx_s_4w	sod_ssxx_2t	sod_ssxx_2x
sod_ssxx_s_2w	sod_ssxx_s_2x	sodhl_asas_4t	sodhl_asas_s_4w
sodhl_sasa_4t	sodhl_sasa_s_4w	sub_2t	sub_leg_x
sub_s_x	sub_t_hh	sub_t_hl	sub_t_lh
sub_t_ll	sub_x	suba_lin	and_2x
and_x	bit_colpsh_sh_w	bit_colpsh_sh_w_da	bit_dintlv2bi_2l
bit_dintlv2bi_l	bit_dintlv3bi_2l	bit_dintlv3bi_l	bit_dintlv3bi_rev_2l
bit_dintlv3bi_rev_l	bit_expnd_2w	bit_expnd_4w	bit_expnd_8b_hh

Table continues on the next page...

Table 4-29. Intrinsics Supported in Emulation Library (continued)

Intrinsics Supported in Emulation Library			
bit_expnd_8b_hl	bit_expnd_8b_lh	bit_expnd_8b_ll	bit_expnd_rev_8b_hh
bit_expnd_rev_8b_hl	bit_expnd_rev_8b_lh	bit_expnd_rev_8b_ll	bit_intlv2bi_2l
bit_intlv2bi_l	bit_intlv3bi_2l	bit_intlv3bi_l	bit_intlv3bi_rev_2l
bit_intlv3bi_rev_l	bit_rev_l	bmchga_h	bmchga_l
bmclra_h	bmclra_l	bmseta_h	bmseta_l
eor_x	extract_u_x	extract_u_x_imm	extract_x
insert_x	insert_x_imm	l_and_2x	l_extract_u_x
l_extract_u_x_imm	l_extract_x	l_extract_x_imm	l_insert_x
l_insert_x_imm	l_nand_2x	l_neor_2x	l_nor_2x
l_nor_x	l_not_2x	l_or_2x	nand_2x
neor_2x	nor_2x	nor_x	not_2x
or_2x	or_x	bmtsta_c_h	bmtsta_s_h
cmp_eq_x	cmp_ge_u_x	cmp_ge_x	cmp_gt_u_x
cmp_gt_x	cmp_le_x	cmp_ne_x	cmpd_eq_2t
cmpd_eq_2w	cmpd_eq_2x	cmpd_eq_4t	cmpd_eq_4w
cmpd_eq_x	cmpd_ge_2t	cmpd_ge_2w	cmpd_ge_2x
cmpd_ge_4t	cmpd_ge_4w	cmpd_ge_x	cmpd_gt_2t
cmpd_gt_2w	cmpd_gt_2x	cmpd_gt_4t	cmpd_gt_4w
cmpd_gt_x	cmpd_le_x	cmpd_lt_x	cmpd_ne_2t
cmpd_ne_2w	cmpd_ne_2x	cmpd_ne_4t	cmpd_ne_4w
cmpd_ne_x	l_cmpd_eq_2x	l_cmpd_eq_x	l_cmpd_ge_2x
l_cmpd_ge_x	l_cmpd_gt_2x	l_cmpd_gt_x	l_cmpd_le_x
l_cmpd_lt_x	l_cmpd_ne_2x	l_cmpd_ne_x	tstbm_c_l
tstbm_s_l	l_ld2_16b	l_ld2_16bf	l_ld2_16f
l_ld2_2b	l_ld2_2bf	l_ld2_2f	l_ld2_32f
l_ld2_4b	l_ld2_4bf	l_ld2_4f	l_ld2_8b
l_ld2_8bf	l_ld2_8f	l_ld2_u_16b	l_ld2_u_2b
l_ld2_u_4b	l_ld2_u_8b	l_ldsh2_dn_2f	l_ldsh2_up_2f
l_st_srs_2bf	l_st_srs_2f	l_st_srs_2l	l_st_srs_4bf
l_st_srs_4f	l_st_srs_4l	l_st_srs_8bf	l_st_srs_8f
l_st_srs_8l	l_st_srs_bf	l_st_srs_f	l_st_srs_l
l_st2_16b	l_st2_16bf	l_st2_2b	l_st2_2bf
l_st2_4b	l_st2_4bf	l_st2_8b	l_st2_8bf
l_st2_srs_16bf	l_st2_srs_16f	l_st2_srs_2bf	l_st2_srs_2f
l_st2_srs_4bf	l_st2_srs_4f	l_st2_srs_8bf	l_st2_srs_8f
ld_16l	ld_2b	ld_2bf	ld_2f
ld_2l	ld_2w	ld_2x	ld_4b
ld_4bf	ld_4f	ld_4l	ld_4w
ld_4x	ld_8b	ld_8bf	ld_8f

Table continues on the next page...

Table 4-29. Intrinsics Supported in Emulation Library (continued)

Intrinsics Supported in Emulation Library			
ld_8l	ld_8w	ld_8x	ld_b
ld_bf	ld_f	ld_l	ld_u_2b
ld_u_2l	ld_u_2w	ld_u_4b	ld_u_4l
ld_u_4w	ld_u_8b	ld_u_8l	ld_u_8w
ld_u_b	ld_u_l	ld_u_w	ld_w
ld_x	ld2_16b	ld2_16bf	ld2_16f
ld2_2b	ld2_2bf	ld2_2f	ld2_32f
ld2_4b	ld2_4bf	ld2_4f	ld2_8b
ld2_8bf	ld2_8f	ld2_u_16b	ld2_u_2b
ld2_u_4b	ld2_u_8b	ldsh2_dn_2f	ldsh2_up_2f
st_16l	st_2b	st_2bf	st_2f
st_2l	st_2w	st_2x	st_4b
st_4bf	st_4f	st_4l	st_4w
st_4x	st_8b	st_8bf	st_8f
st_8l	st_8w	st_8x	st_b
st_bf	st_f	st_l	st_srs_2bf
st_srs_2f	st_srs_2l	st_srs_4bf	st_srs_4f
st_srs_4l	st_srs_8bf	st_srs_8f	st_srs_8l
st_srs_bf	st_srs_f	st_srs_l	st_w
st_x	st2_16b	st2_16bf	st2_2b
st2_2bf	st2_4b	st2_4bf	st2_8b
st2_8bf	st2_srs_16bf	st2_srs_16f	st2_srs_2bf
st2_srs_2f	st2_srs_4bf	st2_srs_4f	st2_srs_8bf
st2_srs_8f	l_mac_2t	l_mac_2t_m	l_mac_2x
l_mac_h_2x	l_mac_h_s_2x	l_mac_hh_2x	l_mac_hh_s_2x
l_mac_i_x_ll	l_mac_i_x_m_ll	l_mac_l_2x	l_mac_l_s_2x
l_mac_leg_x_hh	l_mac_leg_x_hl	l_mac_leg_x_ll	l_mac_leg_x_m_hh
l_mac_leg_x_m_hl	l_mac_leg_x_m_ll	l_mac_ll_2x	l_mac_ll_i_2x
l_mac_ll_s_2x	l_mac_r_2t	l_mac_r_2t_m	l_mac_r_4t
l_mac_r_x_hh	l_mac_r_x_lh	l_mac_r_x_ll	l_mac_rleg_x_hh
l_mac_rleg_x_lh	l_mac_rleg_x_ll	l_mac_rleg_x_m_hh	l_mac_rleg_x_m_lh
l_mac_rleg_x_m_ll	l_mac_s_2x	l_mac_s_x_hh	l_mac_s_x_hl
l_mac_s_x_ll	l_mac_sr_x_hh	l_mac_sr_x_lh	l_mac_sr_x_ll
l_mac_su_i_2t	l_mac_us_i_x_ll	l_mac_uu_i_x_ll	l_mac_uu_i_x_m_ll
l_mac_x_hh	l_mac_x_hl	l_mac_x_ll	l_mac32_r_2x
l_mac32_r_2x_da	l_mac32_r_x	l_mac32_sr_2x	l_mac32_sr_2x_da
l_mac32_sr_x	l_macem32_ss_x_hh	l_macem32_su_x_hl	l_macem32_us_i_x_lh
l_macem32_us_x_lh	l_macem32_uu_i_x_lh	l_macem32_uu_x_ll	l_macem32_xsu_x
l_macem32_xsu_x_m	l_macm_r_2x	l_macm_r_2x_h	l_macm_r_2x_l

Table continues on the next page...

Table 4-29. Intrinsics Supported in Emulation Library (continued)

Intrinsics Supported in Emulation Library			
l_macm_r_x_h	l_macm_r_x_l	l_macm_r_x_m_h	l_macm_sr_2x
l_macm_sr_2x_h	l_macm_sr_2x_l	l_macm_sr_x_h	l_macm_sr_x_l
l_macm_su_log2_x_h	l_macm_su_x_h	l_macm_x_h	l_macq_i_x
l_macq_x	l_macshl_s_x_hh	mac_2t	mac_2t_m
mac_2x	mac_h_2x	mac_h_s_2x	mac_hh_2x
mac_hh_s_2x	mac_i_4w	mac_i_x_ll	mac_i_x_m_ll
mac_l_2x	mac_l_s_2x	mac_leg_x_hh	mac_leg_x_hl
mac_leg_x_ll	mac_leg_x_m_hh	mac_leg_x_m_hl	mac_leg_x_m_ll
mac_ll_2x	mac_ll_i_2x	mac_ll_s_2x	mac_r_2t
mac_r_2t_m	mac_r_4t	mac_r_x_hh	mac_r_x_lh
mac_r_x_ll	mac_rleg_x_hh	mac_rleg_x_lh	mac_rleg_x_ll
mac_rleg_x_m_hh	mac_rleg_x_m_lh	mac_rleg_x_m_ll	mac_s_2w
mac_s_2x	mac_s_x_hh	mac_s_x_hl	mac_s_x_ll
mac_sr_2w	mac_sr_4w	mac_sr_x_hh	mac_sr_x_lh
mac_sr_x_ll	mac_su_i_2t	mac_us_i_x_ll	mac_uu_i_x_ll
mac_uu_i_x_m_ll	mac_x_hh	mac_x_hl	mac_x_ll
mac32_il_l	mac32_il_l_m	mac32_r_2x	mac32_r_2x_da
mac32_r_x	mac32_sr_2x	mac32_sr_2x_da	mac32_sr_x
macem32_ss_x_hh	macem32_su_x_hl	macem32_us_i_x_lh	macem32_us_x_lh
macem32_uu_i_x_lh	macem32_uu_x_ll	macem32_xsu_x	macem32_xsu_x_m
macm_r_2x	macm_r_2x_h	macm_r_2x_l	macm_r_x_h
macm_r_x_l	macm_r_x_m_h	macm_sr_2x	macm_sr_2x_h
macm_sr_2x_l	macm_sr_x_h	macm_sr_x_l	macm_su_log2_x_h
macm_su_x_h	macm_us_il_l_l	macm_x_h	macq_i_x
macq_x	macshl_s_x_hh	l_maccx_2x	l_maccx_2x_m
l_maccx_c_2x	l_maccx_c_2x_m	l_maccx_c_ir_4t	l_maccx_c_r_2t
l_maccx_c_r_4t	l_maccx_c_r_4t_da	l_maccx_c_r_4t_m	l_maccx_c_s_2x
l_maccx_i_2x	l_maccx_ir_4t	l_maccx_is_2x	l_maccx_r_2t
l_maccx_r_4t	l_maccx_r_4t_da	l_maccx_r_4t_m	l_maccx_s_2x
l_maccxd_cpn_2x	l_maccxd_cpn_2x_da	l_maccxd_cpp_2x	l_maccxd_cpp_i_2x
l_maccxd_cpp_ir_2t	l_maccxd_cpp_is_2x	l_maccxd_cpp_r_2t	l_maccxd_cpp_s_2x
l_maccxd_cpp_s_2x_da	l_maccxd_pn_2x	l_maccxd_pn_2x_da	l_maccxd_pn_i_2x
l_maccxd_pn_is_2x	l_maccxd_pn_s_2x	l_maccxd_pn_s_2x_da	l_maccxd_pp_2x
l_maccxd_pp_2x_da	l_maccxd_pp_i_2x	l_maccxd_pp_ir_2t	l_maccxd_pp_is_2x
l_maccxd_pp_r_2t	l_maccxd_pp_s_2x	l_maccxd_pp_s_2x_da	l_maccxd_ppx_2x
l_maccxd_ppx_s_2x	l_maccxm_c_r_2x	l_maccxm_c_r_2x_m	l_maccxm_c_sr_2x
l_maccxm_c_sr_2x_m	l_maccxm_r_2x	l_maccxm_r_2x_m	l_maccxm_sr_2x
l_maccxm_sr_2x_m	maccx_2x	maccx_2x_m	maccx_c_2x
maccx_c_2x_m	maccx_c_ir_4t	maccx_c_isr_4w	maccx_c_r_2t

Table continues on the next page...

Table 4-29. Intrinsics Supported in Emulation Library (continued)

Intrinsics Supported in Emulation Library			
maccx_c_r_4t	maccx_c_r_4t_da	maccx_c_r_4t_m	maccx_c_s_2x
maccx_c_sr_2w	maccx_c_sr_4w	maccx_c_sr_4w_da	maccx_i_2x
maccx_ir_4t	maccx_is_2x	maccx_r_2t	maccx_r_4t
maccx_r_4t_da	maccx_r_4t_m	maccx_s_2x	maccx_sr_2w
maccx_sr_4w	maccx_sr_4w_da	maccxd_cpn_2x	maccxd_cpn_2x_da
maccxd_cpn_s_2x	maccxd_cpn_s_2x_da	maccxd_cpn_sr_2w	maccxd_cpp_2x
maccxd_cpp_2x_da	maccxd_cpp_i_2x	maccxd_cpp_ir_2t	maccxd_cpp_is_2x
maccxd_cpp_isr_2w	maccxd_cpp_r_2t	maccxd_cpp_s_2x	maccxd_cpp_s_2x_da
maccxd_cpp_sr_2w	maccxd_pn_2x	maccxd_pn_2x_da	maccxd_pn_i_2x
maccxd_pn_is_2x	maccxd_pn_isr_2w	maccxd_pn_s_2x	maccxd_pn_s_2x_da
maccxd_pn_sr_2w	maccxd_pp_2x	maccxd_pp_2x_da	maccxd_pp_i_2x
maccxd_pp_ir_2t	maccxd_pp_is_2x	maccxd_pp_isr_2w	maccxd_pp_r_2t
maccxd_pp_s_2x	maccxd_pp_s_2x_da	maccxd_pp_sr_2w	maccxd_ppx_2x
maccxd_ppx_s_2x	maccxd_ppx_sr_2w	maccxm_c_r_2x	maccxm_c_r_2x_m
maccxm_c_sr_2x	maccxm_c_sr_2x_m	maccxm_r_2x	maccxm_r_2x_m
maccxm_sr_2x	maccxm_sr_2x_m	l_macd_2x	l_macd_2x_da
l_macd_2x_m	l_macd_cim_2x	l_macd_cim_i_2x	l_macd_cim_leg_x_m
l_macd_cim_r_2t	l_macd_cim_s_2x	l_macd_cim_s_x	l_macd_cim_x
l_macd_i_2x	l_macd_im_2x	l_macd_im_2x_m	l_macd_im_i_2x
l_macd_im_leg_x	l_macd_im_r_2t	l_macd_im_s_2x	l_macd_im_s_x
l_macd_im_x	l_macd_leg_x	l_macd_leg_x_m	l_macd_r_2t
l_macd_re_2x	l_macd_re_2x_m	l_macd_re_i_2x	l_macd_re_leg_x
l_macd_re_r_2t	l_macd_re_s_2x	l_macd_re_s_x	l_macd_re_x
l_macd_s_2x	l_macd_s_2x_da	l_macd_s_x	l_macd_x
l_macdcf_2x	l_macdcf_h_r_4t	l_macdcf_inv_2x	l_macdcf_invh_r_4t
l_macdcf_invl_r_4t	l_macdcf_l_r_4t	l_macdrf_h_2x	l_macdrf_h_r_2t
l_macdrf_h_r_4t	l_macdrf_h_s_2x	l_macdrf_invh_2x	l_macdrf_invh_r_4t
l_macdrf_invl_2x	l_macdrf_invl_r_4t	l_macdrf_l_2x	l_macdrf_l_r_2t
l_macdrf_l_r_4t	l_macdrf_l_s_2x	macd_2x	macd_2x_da
macd_2x_m	macd_cim_2x	macd_cim_i_2x	macd_cim_leg_x_m
macd_cim_r_2t	macd_cim_s_2x	macd_cim_s_x	macd_cim_sr_2w
macd_cim_x	macd_i_2x	macd_im_2x	macd_im_2x_m
macd_im_i_2x	macd_im_leg_x	macd_im_r_2t	macd_im_s_2x
macd_im_s_x	macd_im_sr_2w	macd_im_x	macd_leg_x
macd_leg_x_m	macd_r_2t	macd_re_2x	macd_re_2x_m
macd_re_i_2x	macd_re_leg_x	macd_re_r_2t	macd_re_s_2x
macd_re_s_x	macd_re_sr_2w	macd_re_x	macd_s_2x
macd_s_2x_da	macd_s_x	macd_sr_2w	macd_x
macdcf_2x	macdcf_h_r_4t	macdcf_h_sr_4w	macdcf_inv_2x

Table continues on the next page...

Table 4-29. Intrinsics Supported in Emulation Library (continued)

Intrinsics Supported in Emulation Library			
macdcf_invh_r_4t	macdcf_invl_r_4t	macdcf_l_r_4t	macdcf_l_sr_4w
macdrf_h_2x	macdrf_h_r_2t	macdrf_h_r_4t	macdrf_h_s_2x
macdrf_h_sr_2w	macdrf_h_sr_4w	macdrf_invh_2x	macdrf_invh_r_4t
macdrf_invl_2x	macdrf_invl_r_4t	macdrf_l_2x	macdrf_l_r_2t
macdrf_l_r_4t	macdrf_l_s_2x	macdrf_l_sr_2w	macdrf_l_sr_4w
l_mpy_2t	l_mpy_2x	l_mpy_h_2x	l_mpy_h_s_2x
l_mpy_hh_2x	l_mpy_hh_s_2x	l_mpy_i_x_ll	l_mpy_l_2x
l_mpy_l_i_4t	l_mpy_l_s_2x	l_mpy_leg_x_hh	l_mpy_leg_x_hl
l_mpy_leg_x_ll	l_mpy_ll_2x	l_mpy_ll_i_2x	l_mpy_ll_s_2x
l_mpy_r_2t	l_mpy_r_4t	l_mpy_r_4t_da	l_mpy_r_x_hh
l_mpy_r_x_lh	l_mpy_r_x_ll	l_mpy_rleg_x_hh	l_mpy_rleg_x_lh
l_mpy_rleg_x_ll	l_mpy_s_2x	l_mpy_s_x_hh	l_mpy_s_x_hl
l_mpy_s_x_ll	l_mpy_su_i_2t	l_mpy_us_i_x_ll	l_mpy_uu_i_x_ll
l_mpy_x_hh	l_mpy_x_hl	l_mpy_x_ll	l_mpy32_pnx_r_2x
l_mpy32_r_2x	l_mpy32_r_x	l_mpy32_sr_2x	l_mpy32_sr_x
l_mpyem32_su_i_x_hl	l_mpyem32_us_x_lh	l_mpyem32_uu_i_x_hl	l_mpyem32_uu_x_ll
l_mpyem32_xsu_x	l_mpym_r_2x	l_mpym_r_2x_h	l_mpym_r_2x_l
l_mpym_r_x_h	l_mpym_r_x_l	l_mpym_sr_2x	l_mpym_sr_2x_h
l_mpym_sr_2x_l	l_mpym_sr_x_h	l_mpym_sr_x_l	l_mpym_x_h
l_mpyq_i_x	l_mpyq_x	l_mpyshl_s_x_hh	mpy_2t
mpy_2x	mpy_h_2x	mpy_h_s_2x	mpy_h_sr_2w
mpy_hh_2x	mpy_hh_s_2x	mpy_hh_sr_2w	mpy_i_x_ll
mpy_l_2x	mpy_l_i_4t	mpy_l_s_2x	mpy_l_sr_2w
mpy_leg_x_hh	mpy_leg_x_hl	mpy_leg_x_ll	mpy_ll_2x
mpy_ll_i_2x	mpy_ll_s_2x	mpy_ll_sr_2w	mpy_r_2t
mpy_r_4t	mpy_r_4t_da	mpy_r_x_hh	mpy_r_x_lh
mpy_r_x_ll	mpy_rleg_x_hh	mpy_rleg_x_lh	mpy_rleg_x_ll
mpy_s_2w	mpy_s_2x	mpy_s_x_hh	mpy_s_x_hl
mpy_s_x_ll	mpy_sr_2w	mpy_sr_4w	mpy_sr_4w_da
mpy_su_i_2t	mpy_us_i_x_ll	mpy_uu_i_x_ll	mpy_x_hh
mpy_x_hl	mpy_x_ll	mpy32_i_ll	mpy32_il_l
mpy32_ll	mpy32_pnx_r_2x	mpy32_r_2x	mpy32_r_x
mpy32_sr_2x	mpy32_sr_x	mpy32_su_i_ll	mpy32_uu_i_ll
mpy32a_i	mpyadda_i	mpyem32_su_i_x_hl	mpyem32_us_x_lh
mpyem32_uu_i_x_hl	mpyem32_uu_x_ll	mpyem32_xsu_x	mpym_i_ll_h
mpym_i_ll_l	mpym_r_2x	mpym_r_2x_h	mpym_r_2x_l
mpym_r_x_h	mpym_r_x_l	mpym_sr_2x	mpym_sr_2x_h
mpym_sr_2x_l	mpym_sr_x_h	mpym_sr_x_l	mpym_us_il_l_l
mpym_x_h	mpyq_i_x	mpyq_x	mpyshl_s_x_hh

Table continues on the next page...

Table 4-29. Intrinsics Supported in Emulation Library (continued)

Intrinsics Supported in Emulation Library			
l_mpycx_2x	l_mpycx_c_2x	l_mpycx_c_r_2t	l_mpycx_c_r_4t
l_mpycx_c_r_4t_da	l_mpycx_c_s_2x	l_mpycx_i_2x	l_mpycx_ir_4t
l_mpycx_r_2t	l_mpycx_r_4t	l_mpycx_r_4t_da	l_mpycx_s_2x
l_mpycxd_cpn_2x	l_mpycxd_cpn_2x_da	l_mpycxd_cpn_s_2x	l_mpycxd_cpn_s_2x_da
l_mpycxd_cpp_2x	l_mpycxd_cpp_2x_da	l_mpycxd_cpp_i_2x	l_mpycxd_cpp_is_2x
l_mpycxd_cpp_s_2x	l_mpycxd_cpp_s_2x_da	l_mpycxd_pn_2x	l_mpycxd_pn_2x_da
l_mpycxd_pn_i_2x	l_mpycxd_pn_is_2x	l_mpycxd_pn_s_2x	l_mpycxd_pn_s_2x_da
l_mpycxd_pp_2x	l_mpycxd_pp_2x_da	l_mpycxd_pp_i_2x	l_mpycxd_pp_is_2x
l_mpycxd_pp_s_2x	l_mpycxd_pp_s_2x_da	l_mpycxm_c_ir_2x	l_mpycxm_c_r_2x
l_mpycxm_c_sr_2x	l_mpycxm_ir_2x	l_mpycxm_r_2x	l_mpycxm_sr_2x
mpycx_2x	mpycx_c_2x	mpycx_c_r_2t	mpycx_c_r_4t
mpycx_c_r_4t_da	mpycx_c_s_2x	mpycx_c_sr_2w	mpycx_c_sr_4w
mpycx_c_sr_4w_da	mpycx_i_2x	mpycx_ir_4t	mpycx_r_2t
mpycx_r_4t	mpycx_r_4t_da	mpycx_s_2x	mpycx_sr_2w
mpycx_sr_4w	mpycx_sr_4w_da	mpycxd_cpn_2x	mpycxd_cpn_2x_da
mpycxd_cpn_s_2x	mpycxd_cpn_s_2x_da	mpycxd_cpn_sr_2w	mpycxd_cpp_2x
mpycxd_cpp_2x_da	mpycxd_cpp_i_2x	mpycxd_cpp_is_2x	mpycxd_cpp_isr_2w
mpycxd_cpp_s_2x	mpycxd_cpp_s_2x_da	mpycxd_cpp_sr_2w	mpycxd_pn_2x
mpycxd_pn_2x_da	mpycxd_pn_i_2x	mpycxd_pn_is_2x	mpycxd_pn_isr_2w
mpycxd_pn_s_2x	mpycxd_pn_s_2x_da	mpycxd_pn_sr_2w	mpycxd_pp_2x
mpycxd_pp_2x_da	mpycxd_pp_i_2x	mpycxd_pp_is_2x	mpycxd_pp_s_2x
mpycxd_pp_s_2x_da	mpycxd_pp_sr_2w	mpycxm_c_ir_2x	mpycxm_c_r_2x
mpycxm_c_sr_2x	mpycxm_ir_2x	mpycxm_r_2x	mpycxm_sr_2x
l_mpyd_2x	l_mpyd_2x_da	l_mpyd_cim_2x	l_mpyd_cim_leg_x_m
l_mpyd_cim_r_2t	l_mpyd_cim_s_2x	l_mpyd_cim_s_x	l_mpyd_cim_x
l_mpyd_im_2x	l_mpyd_im_leg_x	l_mpyd_im_r_2t	l_mpyd_im_s_2x
l_mpyd_im_s_x	l_mpyd_im_x	l_mpyd_leg_x	l_mpyd_leg_x_m
l_mpyd_r_2t	l_mpyd_re_2x	l_mpyd_re_leg_x	l_mpyd_re_r_2t
l_mpyd_re_s_2x	l_mpyd_re_s_x	l_mpyd_re_x	l_mpyd_s_2x
l_mpyd_s_2x_da	l_mpyd_s_x	l_mpyd_x	l_mpydcf_2x
l_mpydcf_h_r_4t	l_mpydcf_inv_2x	l_mpydcf_invh_r_4t	l_mpydcf_invl_r_4t
l_mpydcf_l_r_4t	l_mpydem32_su_i_x	l_mpydem32_uu_i_x	l_mpydrf_h_2x
l_mpydrf_h_r_2t	l_mpydrf_h_r_4t	l_mpydrf_h_s_2x	l_mpydrf_l_2x
l_mpydrf_l_r_2t	l_mpydrf_l_r_4t	l_mpydrf_l_s_2x	mpyd_2x
mpyd_2x_da	mpyd_cim_2x	mpyd_cim_leg_x_m	mpyd_cim_r_2t
mpyd_cim_s_2x	mpyd_cim_s_x	mpyd_cim_sr_2w	mpyd_cim_x
mpyd_im_2x	mpyd_im_leg_x	mpyd_im_r_2t	mpyd_im_s_2x
mpyd_im_s_x	mpyd_im_sr_2w	mpyd_im_x	mpyd_leg_x
mpyd_leg_x_m	mpyd_r_2t	mpyd_re_2x	mpyd_re_leg_x

Table continues on the next page...

Table 4-29. Intrinsics Supported in Emulation Library (continued)

Intrinsics Supported in Emulation Library			
mpyd_re_r_2t	mpyd_re_s_2x	mpyd_re_s_x	mpyd_re_sr_2w
mpyd_re_x	mpyd_s_2x	mpyd_s_2x_da	mpyd_s_x
mpyd_sr_2w	mpyd_x	mpydcf_2x	mpydcf_h_r_4t
mpydcf_inv_2x	mpydcf_invh_r_4t	mpydcf_invl_r_4t	mpydcf_l_r_4t
mpydem32_su_i_x	mpydem32_uu_i_x	mpydrf_h_2x	mpydrf_h_r_2t
mpydrf_h_r_4t	mpydrf_h_s_2x	mpydrf_h_sr_2w	mpydrf_h_sr_4w
mpydrf_l_2x	mpydrf_l_r_2t	mpydrf_l_r_4t	mpydrf_l_s_2x
mpydrf_l_sr_2w	mpydrf_l_sr_4w	avg_u_4b	cast_t_leg_x_h
cast_t_leg_x_l	cast_t_x_h	cast_t_x_l	cast_x_ll
clb_leg_x	clb_lft_x	clba_lft	clip_t_u_2b
clip_w_u_2b	clip_x_u_b	cneg_n_4b	cneg_n_4t
cneg_n_4w	cneg_n_rev_4b	cneg_n_rev_4t	cneg_n_rev_4w
cnegadd_n_2x	cnegadd_n_rev_2x	cob_4b	cob_l
doalign_l	fnd_max_4t	fnd_max_4w	fnd_maxm_4t
fnd_maxm_4w	fnd_min_4t	fnd_min_4w	invsqrt_2x
l_cast_t_leg_x_h	l_cast_t_leg_x_l	l_cast_t_x_h	l_cast_t_x_l
l_clb_leg_x	l_clb_lft_x	l_cnegadd_n_2x	l_cnegadd_n_rev_2x
l_masksel_2x	l_masksel_x	l_pack_f_2t	l_pack_hhhh_4t
l_pack_hhhl_4t	l_pack_hhlh_4t	l_pack_hhll_4t	l_pack_hlhh_4t
l_pack_hlhl_4t	l_pack_hllh_4t	l_pack_hlll_4t	l_pack_l_s_2w
l_pack_l_s_4w	l_pack_lhhh_4t	l_pack_lhhl_4t	l_pack_lhlh_4t
l_pack_lhll_4t	l_pack_llhh_4t	l_pack_llhl_4t	l_pack_lllh_4t
l_pack_t_2t_hh	l_pack_t_2t_hl	l_pack_t_2t_lh	l_pack_t_2t_ll
l_pack_w_2w_hh	l_pack_w_2w_hl	l_pack_w_2w_lh	l_pack_w_2w_ll
l_pack_x_s_2w	l_pack_x_s_4w	l_rnd_leg_x	l_sat_l_w
l_sat_ll_x	l_sat_sc_2f	l_sat_sc_f	l_sat_t_2w
l_sat_t_4w	l_unpack_b_4t	l_unpack_b_u_4t	l_unpack_bf_4t
l_unpack_bh_u_2t	l_unpack_bl_u_2t	l_unpack_thl_2f	l_unpack_thl_2x
l_unpack_tlh_2f	l_unpack_tlh_2x	l_unpack_whl_2f	l_unpack_whl_2x
l_unpack_wlh_2f	l_unpack_wlh_2x	log2	masksel_2x
masksel_x	move_par_w_h	pack_b_4b	pack_bh_8b
pack_bl_8b	pack_f_2t	pack_h_4b	pack_h_8b
pack_hhhh_4t	pack_hhhl_4t	pack_hhlh_4t	pack_hhll_4t
pack_hl_8b	pack_hlhh_4t	pack_hlhl_4t	pack_hllh_4t
pack_hlll_4t	pack_ins0_h3_4w_h	pack_ins0_h3_4w_l	pack_ins0_l3_4w_h
pack_ins0_l3_4w_l	pack_ins1_h3_4w_h	pack_ins1_h3_4w_l	pack_ins2_h3_4w_h
pack_ins2_h3_4w_l	pack_ins3_h3_4w_h	pack_ins3_h3_4w_l	pack_ins3_l3_4w_h
pack_ins3_l3_4w_l	pack_l_4b	pack_l_8b	pack_l_s_2w
pack_l_s_4w	pack_lh_8b	pack_lhhh_4t	pack_lhhl_4t

Table continues on the next page...

Table 4-29. Intrinsics Supported in Emulation Library (continued)

Intrinsics Supported in Emulation Library			
pack_lhlh_4t	pack_lhll_4t	pack_llhh_4t	pack_llhl_4t
pack_lllh_4t	pack_llll_4t	pack_t_2t_hh	pack_t_2t_hl
pack_t_2t_lh	pack_t_2t_ll	pack_w_2w_hh	pack_w_2w_hl
pack_w_2w_lh	pack_w_2w_ll	pack_x_s_2f	pack_x_s_2w
pack_x_s_4f	pack_x_s_4w	recip	rnd_leg_x
sad_4b	sad_8b	sat_l_u_w	sat_l_w
sat_ll_x	sat_sc_2f	sat_sc_f	sat_t_2w
sat_t_4w	sat_t_u_2w	sat_w_2b	sat_x_2l
sat_x_f	sat_x_l	sat_x_u_2w	sat_x_w
sign_4w	swap_dx_4b	swap_dx_8b	swap_ixi_4w
swap_sx_4b	swap_sx_4w	swap_sx_8b	unpack_b_4t
unpack_b_4w	unpack_b_u_4t	unpack_bf_4t	unpack_bh_u_2t
unpack_bl_u_2t	unpack_thl_2f	unpack_thl_2x	unpack_tlh_2f
unpack_tlh_2x	unpack_whl_2f	unpack_whl_2x	unpack_wlh_2f
unpack_wlh_2x	vtrace_l_0_2	vtrace_l_0_3	vtrace_l_0_4
vtrace_l_1_4	vtrace_l_2_4	vtrace_l_3_4	vtrace_l_4_4
vtrace_l_5_4	ash_lft_2l	ash_lft_2t	ash_lft_2w
ash_lft_2x	ash_lft_4t	ash_lft_4w	ash_lft_l
ash_lft_leg_x	ash_lft_ll	ash_lft_s_2l	ash_lft_s_2t
ash_lft_s_2w	ash_lft_s_2x	ash_lft_s_4t	ash_lft_s_4w
ash_lft_s_l	ash_lft_s_x	ash_lft_s20_2t	ash_lft_s20_4t
ash_lft_x	ash_rgt_2l	ash_rgt_2t	ash_rgt_2w
ash_rgt_2x	ash_rgt_4t	ash_rgt_4w	ash_rgt_l
ash_rgt_leg_x	ash_rgt_ll	ash_rgt_s_2l	ash_rgt_s_2t
ash_rgt_s_2w	ash_rgt_s_2x	ash_rgt_s_4t	ash_rgt_s_4w
ash_rgt_s_l	ash_rgt_s_x	ash_rgt_s20_2t	ash_rgt_s20_4t
ash_rgt_x	asha_lft	asha_rgt	l_ash_lft_2x
l_ash_lft_leg_x	l_ash_rgt_2x	l_ash_rgt_leg_x	l_ash_rgt_s_2t
l_ash_rgt_s_4t	lsh_lft_2l	lsh_lft_2w	lsh_lft_2x
lsh_lft_4w	lsh_lft_l	lsh_lft_ll	lsh_lft_x
lsh_rgt_2l	lsh_rgt_2w	lsh_rgt_2x	lsh_rgt_4w
lsh_rgt_l	lsh_rgt_ll	lsh_rgt_x	lsha_lft
lsha_rgt			

4.7.2 Intrinsics supported in MEX library

Table 4-30 lists such intrinsics.

Table 4-30. Intrinsics Supported in MEX Library

Intrinsics Supported in MEX Library		
sc3900__abs_2t	sc3900__abs_2w	sc3900__abs_l
sc3900__abs_s_2w	sc3900__abs_s_l	sc3900__abs_s_x
sc3900__abs_s40_x	sc3900__abs_x	sc3900__absa
sc3900__add_2t	sc3900__add_s_x	sc3900__add_x
sc3900__add_x_imm	sc3900__addla_1_lin	sc3900__addla_2_lin
sc3900__addla_3_lin	sc3900__addla_4_lin	sc3900__addla_5_lin
sc3900__addla_6_lin	sc3900__addm_x_h	sc3900__addm_x_l
sc3900__ash_lft_l	sc3900__ash_lft_s_l	sc3900__ash_lft_s_x
sc3900__ash_lft_x	sc3900__ash_rgt_l	sc3900__ash_rgt_s_l
sc3900__ash_rgt_s_x	sc3900__ash_rgt_x	sc3900__bit_dintlv3bi_l
sc3900__bit_intlv3bi_l	sc3900__bit_rev_4b	sc3900__cmpd_eq_x
sc3900__cmpd_eq_x_imm	sc3900__cmpd_ge_x	sc3900__cmpd_ge_x_imm
sc3900__cmpd_gt_x	sc3900__cmpd_gt_x_imm	sc3900__cmpd_le_x
sc3900__cmpd_lt_x	sc3900__cmpd_ne_x	sc3900__cmpd_ne_x_imm
sc3900__cneg_n_4b	sc3900__Db_get_lsb	sc3900__Db_get_msb
sc3900__extract_x_imm	sc3900__l_abs_s40_x	sc3900__l_add_s_x
sc3900__l_add_x	sc3900__l_cmpd_eq_x	sc3900__l_cmpd_ge_x
sc3900__l_cmpd_gt_x	sc3900__l_cmpd_ne_x	sc3900__l_mac_r_x_hh
sc3900__l_mac_s_x_hh	sc3900__l_mac_s_x_hl	sc3900__l_mac_s_x_ll
sc3900__l_mac_sr_x_hh	sc3900__l_mac_sr_x_lh	sc3900__l_mac_sr_x_ll
sc3900__l_mac_su_i_2t	sc3900__l_mac_x_hh	sc3900__l_mac_x_hl
sc3900__l_mac_x_ll	sc3900__l_macem32_ss_x_hh	sc3900__l_macem32_su_x_hl
sc3900__l_macm_r_x_h	sc3900__l_macm_r_x_l	sc3900__l_macm_r_x_m_h
sc3900__l_macm_sr_x_h	sc3900__l_macm_sr_x_l	sc3900__l_macm_su_log2_x_h
sc3900__l_macm_su_x_h	sc3900__l_macm_x_h	sc3900__l_max_x
sc3900__l_maxm_x	sc3900__l_min_x	sc3900__l_mpy_r_x_hh
sc3900__l_mpy_r_x_lh	sc3900__l_mpy_r_x_ll	sc3900__l_mpy_s_x_hh
sc3900__l_mpy_s_x_hl	sc3900__l_mpy_s_x_ll	sc3900__l_mpy_x_hh
sc3900__l_mpy_x_hl	sc3900__l_mpy_x_ll	sc3900__l_mpydem32_su_i_x
sc3900__l_mpyem32_us_x_lh	sc3900__l_neg_x	sc3900__l_nor_x
sc3900__l_pack_f_2t	sc3900__l_pack_t_2t_hh	sc3900__l_pack_t_2t_hl
sc3900__l_pack_t_2t_lh	sc3900__l_pack_t_2t_ll	sc3900__l_pack_w_2w_hh
sc3900__l_pack_w_2w_hl	sc3900__l_pack_w_2w_lh	sc3900__l_pack_w_2w_ll
sc3900__l_sub_s_x	sc3900__l_sub_x	sc3900__l_to_x
sc3900__lsh_lft_l	sc3900__lsh_rgt_l	sc3900__mac_2t
sc3900__mac_r_x_hh	sc3900__mac_s_x_hh	sc3900__mac_s_x_hl
sc3900__mac_s_x_ll	sc3900__mac_sr_x_hh	sc3900__mac_sr_x_lh
sc3900__mac_sr_x_ll	sc3900__mac_su_i_2t	sc3900__macem32_ss_x_hh

Table continues on the next page...

Table 4-30. Intrinsics Supported in MEX Library (continued)

Intrinsics Supported in MEX Library		
sc3900__macem32_su_x_hl	sc3900__macm_r_x_h	sc3900__macm_r_x_l
sc3900__macm_r_x_m_h	sc3900__macm_sr_x_h	sc3900__macm_sr_x_l
sc3900__macm_su_log2_x_h	sc3900__macm_su_x_h	sc3900__macm_us_il_l_l
sc3900__macm_x_h	sc3900__max_2w	sc3900__max_x
sc3900__maxm_x	sc3900__min_2w	sc3900__min_x
sc3900__mpy_i_x_ll	sc3900__mpy_r_x_hh	sc3900__mpy_r_x_lh
sc3900__mpy_r_x_ll	sc3900__mpy_s_x_hh	sc3900__mpy_s_x_hl
sc3900__mpy_s_x_ll	sc3900__mpy_x_hh	sc3900__mpy_x_hl
sc3900__mpy_x_ll	sc3900__mpyd_x	sc3900__mpydem32_su_i_x
sc3900__mpyem32_us_x_lh	sc3900__neg_x	sc3900__nor_x
sc3900__pack_f_2t	sc3900__pack_l_s_2w	sc3900__pack_t_2t_hh
sc3900__pack_t_2t_hl	sc3900__pack_t_2t_lh	sc3900__pack_t_2t_ll
sc3900__pack_w_2w_hh	sc3900__pack_w_2w_hl	sc3900__pack_w_2w_lh
sc3900__pack_w_2w_ll	sc3900__sat_sc_f	sc3900__sat_t_2w
sc3900__sat_t_u_2w	sc3900__sat_x_f	sc3900__sat_x_l
sc3900__sat_x_w	sc3900__sod_aaii_2t	sc3900__sod_aaii_s_2w
sc3900__sod_aaxx_2t	sc3900__sod_aaxx_s_2w	sc3900__sod_asii_2t
sc3900__sod_asii_s_2w	sc3900__sod_asxx_2t	sc3900__sod_asxx_s_2w
sc3900__sod_saii_2t	sc3900__sod_saii_s_2w	sc3900__sod_saxx_2t
sc3900__sod_saxx_s_2w	sc3900__sod_ssii_2t	sc3900__sod_ssii_s_2w
sc3900__sod_ssxx_2t	sc3900__sod_ssxx_s_2w	sc3900__sub_s_x
sc3900__sub_t_hh	sc3900__sub_t_ll	sc3900__sub_x
sc3900__ux_guard	sc3900__ux_set	sc3900__ux_to_ul
sc3900__x_guard	sc3900__x_set	sc3900__x_to_l

Index

A

B

C

D

E

F

G

H

I

K

L

N

O

P

R

S

T

U

W

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number CWSCCMPREF
Revision 10.9.0, 10/2015